

Delphi

Borland

核心技术丛书



Delphi

面向对象编程思想

刘艺 著

从 **Delphi RAD** 快手成长为 **OOP** 高手的必读秘笈



附赠



机械工业出版社
China Machine Press

Delphi 面向对象编程思想

“选择面向对象编程就意味着你需要抛弃某些可视化编程方法。”

——Marco Cantu

“以非面向对象的方法使用面向对象的工具是一个错误。使用Delphi编写结构化程序可以很快地到达beta版……你的程序可能永远脱离不了beta版。能迅速得到错误的答案，仍然是错误的。”

——Paul Kimmel

“请记住，成功的开发者只需写少量的高质量代码，而不是大量的普通代码。”


——Grady Booch

本书主要内容：

- 面向对象编程入门和 Delphi 的对象模型。
- 理解 Delphi 对象的实质，活用 Delphi 对象的技巧。
- 多态、接口、虚方法、抽象类等概念的剖析和面向对象编程上的具体应用。
- 建立在动态绑定机制上和类型转换机制上的面向对象高级编程技巧。
- 为了实现程序的可维护性、可扩展性和可重用性，而对封装、界面和业务对象的分离、分布式多层体系结构以及实现界面和业务应用跨平台的深入讨论。
- VCL 的内幕资料和研究心得。



光盘包含了
书中绝大多
数示例代码

 网上购书：www.china-pub.com

封面设计 / 江丽萍

ISBN 7-111-12772-2



9 787111 127727



华夏图书

北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259, 68995264
读者服务信箱：hzedu@hzbook.com
www.hzbook.com

ISBN 7-111-12772-2/TP · 2861

定价：55.00 元（附光盘）

Borland 核心技术丛书

Delphi 面向对象编程思想

刘 艺 著

 **机械工业出版社**
China Machine Press

这是一本纯粹讨论 Delphi 面向对象编程的力作。本书以精通 Delphi 面向对象编程为目的,深入浅出地讲解了 Delphi 面向对象的概念和实质、方法和经验、思想和实践;详尽讨论了 Delphi 建立在虚方法、抽象方法、对象接口等动态绑定机制上和向上转型、向下转型、接口转型等类型转换机制上的面向对象高级技巧;并深入研究了通过封装从而实现界面和业务对象的分离,从界面和业务分离逐步实现分布式多层体系结构,进而实现界面和业务应用的跨平台的企业级解决方案。本书还提供了 VCL 的内幕资料和研究心得。

全书使用 Delphi 7 附带的 ModelMaker 实现 UML 对象建模,并附有大量 Delphi 源代码实例,方便读者研究学习。

本书适用于有一定 Delphi 基础,并希望掌握面向对象编程思想和方法,进一步提升水平的软件开发人员。同样,已经掌握面向对象编程的 Java 和 C++ 程序员通过本书亦能快速掌握 Delphi 编程。本书还适合大专院校用于基于 Object Pascal/Delphi 的面向对象编程教学。

版权所有,侵权必究。

图书在版编目 (CIP) 数据

Delphi 面向对象编程思想/刘艺著. — 北京:机械工业出版社, 2003.9
(Borland 核心技术丛书)
ISBN 7-111-12772-2

I .D… II .刘… III .软件工具—程序设计 IV .TP311.56

中国版本图书馆 CIP 数据核字 (2003) 第 068440 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:李云静

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2003 年 9 月第 1 版第 1 次印刷

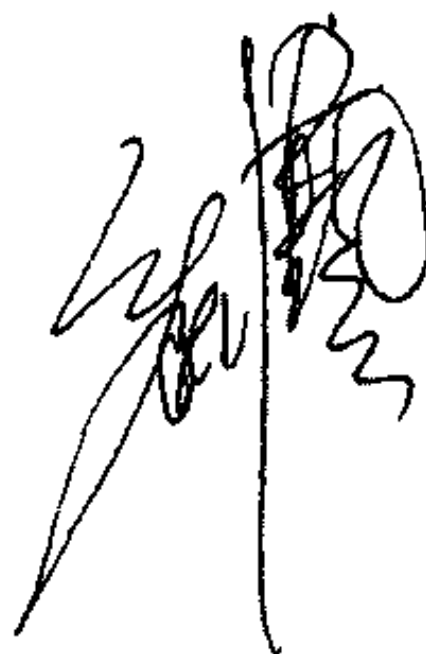
787mm×1092mm 1/16·30.75 印张

印数:0 001-4 000 册

定价:55.00 元 (附光盘)

凡购本书,如有倒页、脱页、缺页,由本社发行部调换
本社购书热线电话:(010) 68326294

这是一本纯粹讨论 Delphi 面向对象编程的书，面向对象不是本书的时髦点缀，而是这本书的核心和全部。

A handwritten signature in black ink, consisting of stylized, overlapping loops and lines, positioned to the right of the main text block.



作者简介：

刘艺，海军工程大学信息与电气学院副教授、美国 Borland 公司授予的 Delphi 产品专家、计算机技术作家。著有《Delphi 6 企业级解决方案及应用剖析》、《Delphi 第三方控件使用大全》等 10 多部计算机专著，出版重点大学计算机教材 2 部。其主持的多个科研项目在军内获奖。

作者个人网站：<http://www.liu-yi.net>

前 言

第一次知道 Delphi 并不是因为 Borland 公司的 Delphi 软件，而是在柏拉图的经典著作《柏拉图对话录》“申辩篇”^①中读到了这个单词：

A friend of mine ... went to Delphi and boldly asked the oracle to tell him whether ... there was anyone wiser than I was, and the Pythian prophetess answered that there was no one wiser ...

Delphi 和苏格拉底的智慧

Delphi 是雅典西北方一座美丽而神圣的小城，传说古时众神之父宙斯测量大地，而 Delphi 正好是世界的中央。《柏拉图对话录》记载了苏格拉底的一位朋友前往 Delphi 向预言女神 Pythian 询问谁是最智慧的人。Pythian 说没有人比苏格拉底更智慧。苏格拉底深感不解，因为他发现自己身边有很多政治家、诗人、哲学家和艺术家。难道这些“专家”和“权威”不是更有智慧吗？自己比起他们差多了。

苏格拉底一一拜访了这些“专家”和“权威”，却发现他们往往自以为是，自欺欺人，对自己不懂的东西也假装知晓。通过拜访，同时苏格拉底也发现自己在许多方面知之甚少。但苏格拉底并没有不懂装懂，他坦诚了自己的无知。这就是预言女神 Pythian 所说的真正的“智慧”。

其实这种智慧不是古希腊人最早提出来的。早在苏格拉底之前，我国的老子就已经总结过了。《道德经》中就有“知人者智，自知者明”，此之谓也。

现在，我们使用的 Delphi 已经是一个优秀的编程语言和软件开发工具了。然而，面对博大精深、不断发展的 Delphi，我们在许多方面还知之甚少。可是在学习和使用 Delphi 时，我们是否也具备了苏格拉底那种自知自明的态度和知所不知的智慧呢？

有许多选择 Delphi 的朋友，最初的想法可能是因为 Delphi 功能强大，易学易用。他们甚至 3 个月就声称精通了 Delphi，半年就敢独立开发软件。其实他们所能做的工作仅仅是控件的拖拉而已。当他们为自己的程序陶醉时，实际上是在为 Delphi 的精巧睿智和别人的控件所陶醉。这种 Delphi 程序员往往被别人戏称为“拖拉”员。而他们却俨然以 Delphi 高手自居。

也有许多放弃 Delphi 的朋友，最初的想法可能是因为觉得 Delphi 只是一个类似 VB 的 RAD 工具。在他们看来 Delphi 就是控件编程，无法像 C++ 或 Java 那样真正实现 OOP。他们怀疑 Delphi 是不是具备面向对象编程语言的特性，能不能实现多态、模式等面向对象的技术。在他们眼里只有使用 C++ 或 Java 的人才是真正的高手。

那么，什么是真正的高手，怎样才能成为一个高手呢？

① 《The Dialogues of Plato》之“Apology”。

论“器”与“气”

首先让我们来看一看什么是武林高手，或许我们能够从中得到启发。

凡是喜爱武功或武侠小说的人都知道，修炼武功分为外练和内修两种途径。外练拳脚、兵器，拳脚的招式和兵器的好坏是关键；内修真气，练精化气、练气化神、练神还虚是根本。这两种修炼方法的最大差别在于时间和功力的函数关系上，如图1所示。通俗地讲，头3年，练外功的可以轻易打败练内功的；第10年，双方只能打个平手；15年后，无论你怎么练外功都不是练内功的对手。20年后，内功高手天下无敌。其中的道理正是在于“器”和“气”的辩证关系。

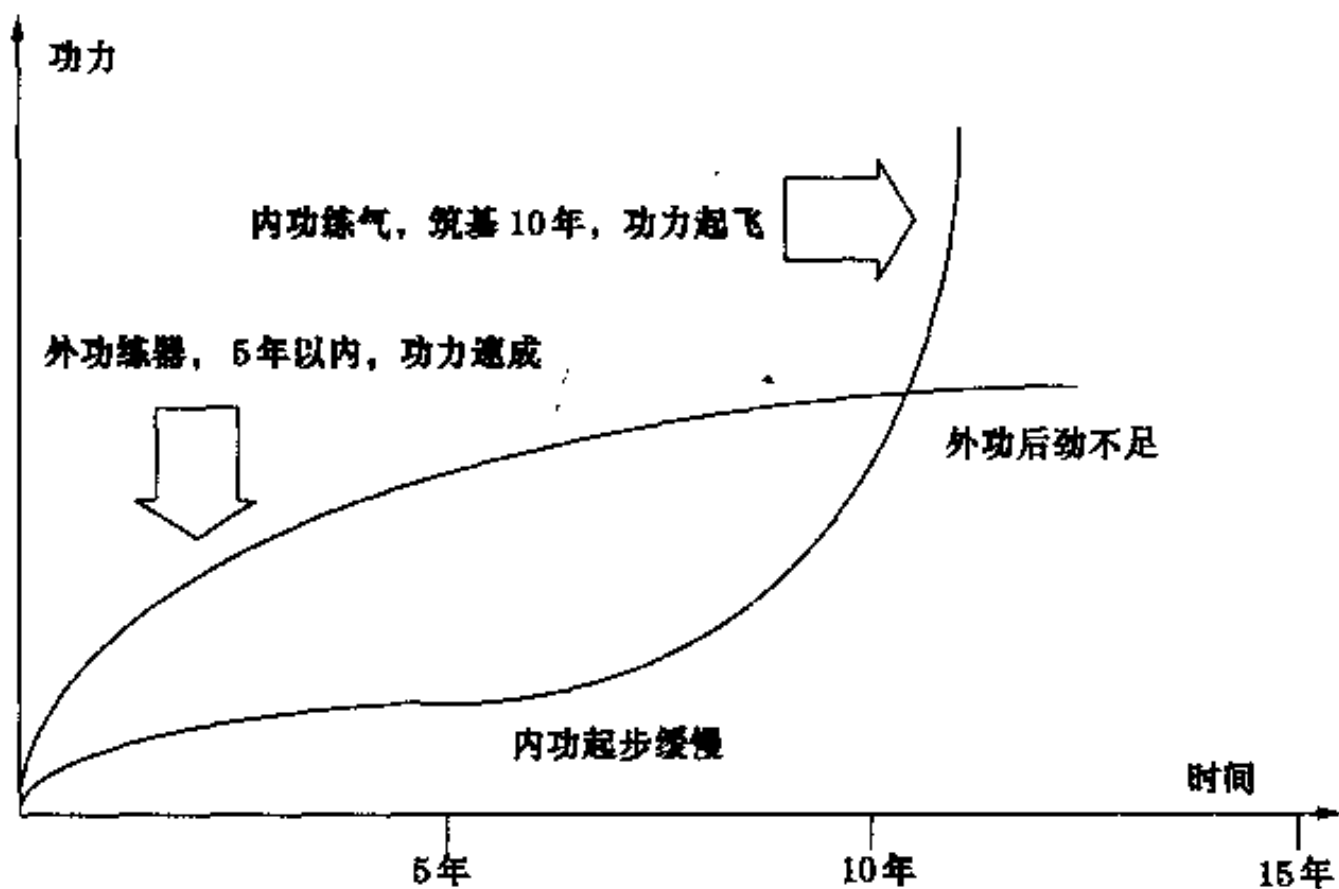


图1 两种修炼方法的时间和功力函数图

“器”为有形之物，刀枪剑戟皆为有形之器。外练武功，离不开这些有形之器，练功即为练兵器。所以使枪弄剑的武林高手往往依赖于兵器的好坏。

“气”为无形之质，一切智慧、法力、思想皆为无形之质。内修气功，离不开这些无形之质，练功即为练气。所以真正的武林高手往往无需依赖于特定的兵器，对于他们而言，任何有形之物皆可随气所运，拟为兵器。一折扇，一拂尘在其手中威力不亚于名枪好剑，是得气所致也。

在软件开发中，编程工具是“器”，编程思想是“气”。

在编程之器中，易用之器如VB、PB；难用之器如汇编、C++；古老之器如FORTRAN、COBOL；时尚之器如Java、C#。

在编程思想中，又有面向对象和面向过程之分，它们既是世界观又是方法论。前者反映的是人对客观对象的思维方式，后者反映的是机器对指令的思维方式。在软件开发的不断实践中，前者的优越性已经得到不断的体现和证实。

掌握面向对象的编程思想如同获得练气的真谛。它的重要性往往胜过了对编程语言的选择!

有人即使选择了面向对象的利器,也无法成为真正的高手,因为他看重的是“器”的好坏,忽略了“气”的修炼。

实际上“一个系统或语言是不是面向对象的并不重要,重要的是怎样才能是面向对象的以及用什么办法实现相关的好处”(《面向对象方法:原理与实践》机械工业出版社 2003 年 3 月出版)。

练器虽易,但难成高手。练气虽好,但见效缓慢,寂寞难耐,非一般常人可以明心见性,直取大道。所以很多武林高手都是先练器、后练气;内外兼修,终成正果。

对于初入武林的新手,他们需要借助兵器的威力,以补内力之不足,器的好坏往往很重要。但随着武功的增加,内力的勃发,对器的依赖性应该减小。《神雕侠侣》中杨过练剑,起步初学时好使利剑,谙悉剑法后喜用钝剑,内功纯熟后树枝亦当剑。所以对于真正的高手而言,剑器的好坏往往并不重要。

同样,软件高手的成长也有这样的过程。初学编程需要选择好的语言,这样可以取得事半功倍的效果,同时激发学习兴趣,增强信心。一旦熟悉了一种语言之后,应以此为契机进而掌握面向对象编程思想。这时你熟悉的不再是语言本身的语法、函数、类库,而是绑定、多态、模式等思维方法,然后触类旁通,再学其他面向对象语言也不难。苦练内功,勇于实践,最终成功的真正的软件高手,是不受编程语言限制的。他们可能比较熟悉一种开发工具,但那也只是承载他们大道无形之气的器具。他们虚心好学,善于总结。他们的思想、方法、模式甚至哲学,既超越了编程语言,又可以指导编程语言的实践。

Delphi 为软件高手的成长提供了内外兼修的捷径。学练 Delphi,既可用其 RAD 之长,控件之利,初学起步,迅速击败对手;学练 Delphi,也能以其 OOP 之能, VCL 之强,培根固本,成就不败之功。

威震四海的华山剑派曾分为“剑宗”和“气宗”,前者只练器,讲招式;后者兼练气,重筑基。学习 Delphi 好比修炼华山剑法,走 RAD 之路是“剑宗”,从 OOP 之道是“气宗”。前者喜用控件,看中奇技淫巧;后者好为对象,热衷方法模式。前者追求速成,后者志存高远。

我认为,无论是为 RAD 而选择 Delphi 还是因 OOP 而放弃 Delphi 的朋友都没有真正了解 Delphi。Delphi 是一个不错的 RAD 开发软件,可是不学 OOP,不深入 VCL 就很难成为真正的高手。同样,Delphi 是一个地道的 OOP 编程工具,结合 Delphi 强大的 RAD 和高效的编译器,可以比其他 OOP 语言有更多的优势和更高的效率。如果能打破门户之见,“器”与“气”同练,内外兼修,我相信 Delphi 程序员不难从一个 RAD 快手成长为 OOP 高手,最终笑傲江湖,纵横四海。

面向对象编程思想和大道无形之气

前面我简单讨论了“器”与“气”的辩证关系。在编程中,修持内功、提高内力的关键之一在于掌握面向对象编程思想。实际上,我更认为面向对象编程思想才最合大道无形之真气的

妙处。

为什么这么讲?“古之大化者,乃与无形俱生”(《鬼谷子》反应第二),气的奥妙首先在于它的“大化”。大化者,天地之大造化也,集一切创造和变化之能。面向对象的编程思想具备了这种特质。

老子曰“无名天地之始,有名万物之母”,无名是无以名状,无法定义的意思。所谓面向对象的思维方式,最奥妙之处在于如何从“无名”中识别和定义对象,如何从“有名”中构造和使用对象。

对于软件开发人员来说,认识客观实体的过程、对用户需求进行分析和设计的过程,就是发现和界定对象的过程,是从无名到有名的过程。然而这里的对象又不同于面向过程中的变量或函数,对象是由类创建的,类是概念的抽象,是可以定义的“有名”,是对象之母。

于是乎,太极生两仪,两仪生八卦,通过类的继承和派生,万物始生,系统构成。

即使作为面向对象编程工具的“器”,也体现和承载了面向对象的编程思想之“气”。

号称“万古丹经王”的内功练气经典之作《周易参同契》开篇第一句话就是“乾坤者,易之门户,众卦之父母。坎离匡郭,运轂正轴,牝牡四卦,以为橐籥。”

从软件开发的角来理解,面向对象的编程工具虽然提供了构建无穷种软件系统的可能性,但这种无限性却是建立在自身类库的有限框架之中。无论是 Delphi 的 VCL,还是 Java 的类库,或是 .NET 框架,无一不是建立在这样一个类似于周易八卦的架构之中。“易有太极,太极生两仪,两仪生四象,四象生八卦”,这样的结构完美无瑕,有着无穷的创造力。

太极是 Delphi 中的 TObject,它是构建系统的原子,并是所有类的祖先,它具有所有类的基本特征。在 Delphi 的编程世界中,根类 TObject 生持久对象类 TPersistent,持久对象类 TPersistent 生组件对象类 TComponent,进而为开发应用程序提供了丰富的控件和强大的功能。

然而类库的结构框架不仅仅是给我们可以作为“器”用的组件,更重要的是这种结构通过类之间的关系和相关作用,实现了“气”的构造和变化,体现出面向对象编程思想的精髓。为我们创建自己的系统提供了绝好的示范。

气的奥妙其次在于它“生于无形”。无形意味着它的自由性、开放性、适应性。在面向对象编程思想中处处充满着“气”的这种大道无形的智慧。

比如,面向编程对象中的多态性使程序员可以撰写更加通用的、更加开放的程序。程序员可以为 Vehicle 对象编写一个纯虚抽象方法 stop(),这样的通用 stop()方法与驾驶什么车无关。程序员可以让派生类去操心如何完成 stop()方法,而继续在更高的抽象级别上编写自己的通用过程。即使 Car 对象的 stop()方法与 Bicycle 对象的 stop()方法完全不一样,程序员也可以使用 Vehicle.stop()。多态性可以让创建的对象自动知道哪一个合适的方法将被调用。这就使程序具备了“气”的开放性和适应性。

练气功中讲究“上德无为,不以察求。下德为之,其用不体”(引自《周易参同契》)。

在面向对象编程思想中,“上德”是晚绑定的纯虚抽象方法,是以不变应万变的对象接口,它是对事物的高度抽象,是形而上学。“上德无为”是说在抽象层次,通过无为来体现编程“虚”的一面,因为这时还无法确定实际使用的真正对象(可能是 Car 对象也可能是 Bicycle 对

象,更可能是以后发明的新交通工具对象),“不以察求”要求我们跳出具体需求的约束,不去考虑具体的实现代码。所以纯虚抽象方法或对象接口中是没有任何代码实现的。

在面向对象编程思想中,“下德”是对纯虚抽象方法的覆盖,是对对象接口的实现。“下德为之”,提供了真正的代码实现。“其用不体”,满足了不断变化的需求。

多态性使得程序员在以后不费多大力气就可以派生对象,实现程序。假设程序员在为 Car 和 Bicycle 构建应用程序,并不知道还存在 Truck,但这并不要紧。程序员可以为继承 Vehicle 类的 Car 和 Bicycle 类撰写覆盖 stop()的方法。这样,在程序中只要把创建的 Car 和 Bicycle 对象转型为 Vehicle 的类型,使用 Vehicle 的 stop()方法,就可以让 Car 和 Bicycle 对象动态绑定符合自己要求的 stop()方法。即使后来新增了 Truck 对象,仍然是调用 Vehicle 的 stop()方法,并不需要改动程序。

“物有自然,事有合离。有近而不可见,有远而可知。近而不可见者,不察其辞也;远而可知者,反往以验来也。”(《鬼谷子》抵戏第四)

虽然客观事物复杂,用户需求是变化的,但是其中也有一定的内在规律。

近而不可见者,只看眼前的具体功能实现,不察事物的一般发展规律,心中只有孤立的数据和机械的流程,一旦“事有合离”,则措手不及,难以应对。这样的编程是静态的、机械的、难以维护和扩展的。

远而可知者,善于发现规律,重视代码重用,眼中尽是有机的对象和和谐的关系,即使需求改变,也能从容应对,游刃有余。这样的编程是动态的、灵活的、可维护和可扩展的。

Paul Kimmel 在《Delphi 6 应用开发指南》中说“以非面向对象的方法去使用面向对象工具是一个错误。使用 Delphi 编写结构化程序可以很快地到达 beta 版……你的程序可能永远脱离不了 beta 版。迅速得到错误的答案,仍然是错误的。”

同样是使用 Delphi,如果没有面向对象的编程思想,好比“不察其辞”,最终仍然是“近而不可见”,难以开发出优秀的系统。惟有潜心苦练,悉心总结,掌握面向对象编程思想的精髓,才能运“气”自如,“反往以验来也”,最终达到“远而可知”的境界。

关于本书

从第一个真正面向对象的语言 Smalltalk (1972 年) 出现至今已经有 30 多年的历史了。然而书店中充斥着的面向对象编程的书籍大都是 C++ 和 Java,似乎面向对象的语言仅有这两种,而实际上真正的面向对象语言却有 4 个基本分支,近 20 种之多。由于 Delphi 面向对象编程的书籍很少,不少程序员为了学习 OOP,不得不放弃 Delphi。这真是 Delphi 的一大悲哀。当我读到 Bruce Eckel 的《Thinking in Java》,就感叹过为什么就没有这样的 Delphi 大作呢?

其实,Delphi 系出名门,它是 Borland 公司在 Object Pascal 基础上开发的。现在, Borland 公司从 Delphi 7 开始使用 Delphi 语言来取代 Object Pascal 叫法^①。Delphi 在 OOP 方面实际上并不比 C++ 和 Java 逊色,这一点读者可以参见本书附录 B “面向对象编程语言比较: Java、C++ 和

① 参见 Delphi 帮助文件“Delphi Language Reference”。在本书中笔者也将 Object Pascal 不加区别地称为 Delphi。

Delphi”。

为此，我一直打算写一本 Delphi 面向对象编程的书，总结自己在 Delphi 面向对象编程方面的学习体会和实践经验。然而这是一项难度不小的工作，全书从构思到下笔花费了很长的时间，直到今年 6 月才算正式完稿。刚巧今年也是 Borland 创建 20 周年，作为 Borland 产品 Delphi 的用户，拙作的出版也算是对此的纪念。

这是一本纯粹讨论 Delphi 面向对象编程的书，面向对象不是本书的时髦点缀，而是这本书的核心和全部。

本书自第 1 章“建立面向对象的新思维”开始就试图从面向对象编程的历史和现状入手，阐述面向对象编程思想的起源发展和基本观念，以及面向对象建模方法和 UML 的应用。这一章是为了帮助读者建立面向对象的基本概念，了解面向对象的思维方法。

第 2 章“Delphi 对象模型”介绍了 Delphi 面向对象编程的基础知识及其对象模型结构体系。

第 3 章“理解对象”从对象的本质、生死、关系三方面深入讨论了对象的内部机制、生命周期、相互作用，为读者了解和掌握对象打下了基础。

第 4 章“使用对象”讲解了在 Delphi 面向对象编程中如何高效使用对象。这里重点讨论了界面对象、组件对象、对象集和对象参数的使用方法和技巧，并对 VCL 组件使用和开发中的常见问题进行了深入思考。

第 5 章“深入多态”介绍了多态的概念及其在编程中的应用。其中通过大量的实例讲解了重载和覆盖、虚方法与动态方法、抽象类和抽象方法、类的类型转换等重要概念和思维方法。

第 6 章“剖析接口”全面介绍了对象接口的编程知识和应用技巧。阐述了接口在实现动态绑定和多重继承方面的重要作用，演示了接口在面向对象编程中的实际用法。

第 7 章“研究封装”阐明了封装在面向对象编程中的重要意义和应用原则，并分别从逻辑上的封装和物理上的封装来进一步讨论封装的实现方法和应用技巧。

第 8 章“实现界面和业务的分离”将面向对象编程应用到一个新的高度。这一章通过界面和业务分离的演化实例，讲解了如何利用面向对象的设计将一个桌面程序进化到分布式多层系统。并结合 Delphi 的最新 Web 技术，介绍了如何用 Web Service 封装业务对象，用 Web Form 封装界面对象，用新技术封装旧对象，从而实现跨平台的应用。

最后本书第 9 章和第 10 章“深入浅出 VCL”，研究了 VCL 的内部机制，并剖析了 VCL 重要类系的对象用法，为渴望深入提高编程水平的读者提供了参考。

从本书的组成结构上看可以划分成五大部分。

第一部分，全书的前两章是 Delphi 面向对象编程的入门。已经掌握面向对象基本概念并有 Delphi 编程经验的读者可以跳过这两章。

第二部分，第 3、4 章是 Delphi 面向对象编程的关键。不掌握对象的实质，就无法使用好对象。

第三部分，第 5、6 章是 Delphi 面向对象编程的深入。面向对象的高级技巧无一不是建立在虚方法、抽象方法、对象接口等动态绑定机制上和向上转型、向下转型、接口转型等类型转换机制上的。

第四部分，第 7、8 章是 Delphi 面向对象编程的应用。为了实现程序的可维护性、可扩展性和可重用性，封装已经成为面向对象编程的重要思想之一。通过封装从而实现界面和业务对象的分离，从界面和业务分离逐步实现分布式多层体系结构，进而实现界面和业务应用的跨平台。这里演示了基于面向对象编程思想的从一般应用程序到企业级应用程序的解决方案。

第五部分，最后的第 9 章和第 10 章是 Delphi 面向对象编程的参考。熟悉 VCL、学习 VCL 对精通 Delphi 十分有帮助。鉴于目前 VCL 内幕资料的缺乏，因而这一部分所提供的内容可能比较有限，但却是很难得的。

准确地讲，这本书不是写给“高手”的，而是写给那些想从 RAD 向 OOP 转变的程序员的，以及希望通过 Delphi 来学习 OOP 的朋友的。我觉得作为一本比较实用的中级 Delphi 技术书，比较合适。所以在全书的行文中，力求通俗易懂，图文并茂，并精心编写了大量的示例程序（随书光盘源代码超过 50MB），供读者研习。这本书的核心是 OOP，而不是针对 Delphi 的所有方面。阅读本书需要有一定的 Delphi 基础，书中涉及到一些专门的知识（如：COM+ 等），还需读者进一步参阅相关书籍。

可能会有一些“高手”对本书失望。我觉得自己不适合写“高手”阅读的书，因为我就不是高手，我觉得自己永远是新手。和其他新手不同的是，我使用 Delphi 的时间更长一点，经验和阅历稍多一点。所以，若发现本书有不妥之处敬请读者指正，不尽如人意之处希望多多包涵。

网友 xzh2000 讲得好：“一本书的生命很重要，如果作者能花心血经常修改补充，才能成就经典！”的确一本好书需要经过多次反复修订才能成为经典之作，所以我愿意聆听所有读者的宝贵建议，并希望这本书能够不断修订再版。

光盘内容

本书光盘包含了书中绝大多数示例代码。

光盘内容按章编排。其运行环境为 Windows 98 以上的 Windows 操作系统，完全安装需要不少于 70MB 的硬盘空间。

光盘提供了完整的 Delphi 项目文件，可用 Delphi 直接打开。所有项目文件和源程序均在 Delphi 7 上调试通过。

致谢

其实要想获得 Delphi 预言女神 Pythian 所说的“智慧”并不是一件简单的事。随着我们不断地学习，不断地获得经验，我们的感觉通常是自己知道的越来越多。但实际上，因为我们知识水平的提高和认识能力的增加，反而更能发现自己不知道的领域，意识到自己的无知。

每当拙作完稿之际，我总是发现自己仅仅写了一些皮毛。虽然通过自己的体会和经验解决了一些问题，但又有更多的问题涌现，有待研究。于是自己像一个新手一样又要重新开始学习。

当然，一本书的写作离不开众人的智慧。在这本书的写作中我参考了大量的资料，包括

Delphi 以外的很多资料。在书后的参考文献中我列出了主要的参考资料，并向这些资料的作者表示敬意。

在此特别要感谢的是胡冰乐先生，他无私地将自己撰写和整理的 VCL 相关资料提供给我参考。

参加本书资料整理、审稿校对、文字录入工作的人员有：段立、李启元、罗勇、吴苗、王剑、刘卫华、尹亚兰、洪蕾、吴英等。为本书撰写提供其他帮助的还有：王永斌、周安栋、屈晓旭、曹旭峰，在此一并表示感谢。

最后感谢 Borland 北京代表处王玉红女士的大力支持。

刘 艺

<http://www.liu-yi.net>

E-mail: my_reader@sina.com

2003 年 6 月 12 日于南京

目 录

前言

第 1 章 建立面向对象的新思维	1
1.1 导论	1
1.1.1 历史背景	1
1.1.2 面向过程和面向对象	4
1.1.3 面向对象的技术背景和特点	6
1.1.4 为什么要使用面向对象的编程 技术	8
1.2 面向对象的基本概念	10
1.2.1 类和对象	10
1.2.2 封装	12
1.2.3 继承	12
1.2.4 多态性	13
1.3 面向对象建模和 UML	14
1.3.1 面向对象建模	14
1.3.2 UML 是什么	15
1.3.3 Delphi 面向对象建模工具 ModelMaker	17
1.3.4 UML 建模示例 (ModelMaker 实现)	20
第 2 章 Delphi 对象模型	31
2.1 类和对象	31
2.1.1 类	31
2.1.2 类成员	32
2.1.3 对象	33
2.1.4 类操作符	34
2.2 方法	35
2.2.1 什么是方法	35
2.2.2 方法的分类	36
2.2.3 方法的绑定机制	40
2.3 可见性	45
2.4 属性	46
2.4.1 什么是属性	46

2.4.2 使用数组属性	48
2.5 异常	49
2.5.1 异常是一种特殊的对象	49
2.5.2 如何捕捉和处理异常	50
第 3 章 理解对象	53
3.1 对象的本质	53
3.1.1 什么是对象	53
3.1.2 对象在哪里	55
3.1.3 对象引用和类引用	59
3.1.4 对象的传递	63
3.1.5 对象的克隆	65
3.2 对象的生死	69
3.2.1 对象的构造和析构	69
3.2.2 如何动态生成对象	74
3.2.3 对象的生命期	81
3.2.4 组件对象生命期管理的误区	81
3.3 对象的关系	87
3.3.1 对象、类和类型	89
3.3.2 对象之间的关系基础	91
3.3.3 对象的继承与合成	92
3.3.4 依赖关系和合作关系	107
第 4 章 使用对象	117
4.1 应用程序和界面对象	117
4.1.1 Windows 应用程序和 Application 对象	117
4.1.2 窗体和对话框	118
4.1.3 界面对象和 UI 框架	123
4.2 使用 VCL 组件对象	128
4.2.1 组件和控件	128
4.2.2 组件对象使用实例	130
4.2.3 组件使用的误区	140
4.3 使用对象集	143
4.3.1 对象数组	143

4.3.2 容器对象	151	6.7 接口的其他用法探索	263
4.4 使用对象参数	163	第 7 章 研究封装	271
4.5 组件开发中的面向对象思考	173	7.1 什么是封装	271
4.5.1 开发 VCL 组件	173	7.1.1 封装的概念	271
4.5.2 继承	175	7.1.2 切割和封装的原则	272
4.5.3 合成与嵌入	180	7.2 逻辑上的封装	274
4.5.4 链接	184	7.2.1 类的封装	274
第 5 章 深入多态	187	7.2.2 数据的封装	278
5.1 认识多态	187	7.3 物理上的封装	290
5.2 重载与覆盖	188	7.3.1 物理封装和动态链接	290
5.2.1 重载	188	7.3.2 用 DLL 封装对象	294
5.2.2 覆盖	189	7.3.3 用 COM/COM+ 封装对象	302
5.3 虚方法与动态方法	196	第 8 章 实现界面和业务的分离	311
5.4 抽象类与抽象方法	197	8.1 关于界面和业务的分离	311
5.5 类的类型转换	200	8.1.1 从封装到界面和业务分离	311
5.5.1 向上转型	201	8.1.2 从界面和业务分离到分布式多层 体系结构	312
5.5.2 向下转型	202	8.2 界面和业务分离的演化实例	314
5.6 多态和面向对象编程	208	8.2.1 一个典型的 RAD 程序	314
5.7 用 VCL 的抽象类实现多态	211	8.2.2 界面和业务的逻辑分离	318
第 6 章 剖析接口	217	8.2.3 界面和业务的物理分离	325
6.1 认识接口	217	8.2.4 界面和业务的物理分离	330
6.1.1 什么是接口	217	8.3 Web Service: 实现业务跨平台	338
6.1.2 使用对象	217	8.3.1 Web Service 是一种部署在 Web 上 的对象	338
6.1.3 接口的引入	218	8.3.2 创建 SOAP Server 应用程序	340
6.1.4 接口和多态性	220	8.3.3 用 Web Service 封装业务对象	342
6.2 使用接口	220	8.3.4 创建调用 Web Service 的客户端 程序	350
6.2.1 定义接口	221	8.3.5 Web Service 类型的转换和部署	354
6.2.2 实现接口	222	8.4 Web Form: 实现界面跨平台	361
6.3 接口与抽象类	230	8.4.1 IntraWeb: Delphi 的 Web Form 解决方案	361
6.4 接口关系	235	8.4.2 创建一个 Web Form 程序	362
6.4.1 类、对象和接口的关系	235	8.4.3 IntraWeb 和业务对象整合	371
6.4.2 接口引用关系	236	8.4.4 IntraWeb 和 Web Service 整合	379
6.4.3 互相依赖的接口	237	第 9 章 深入浅出 VCL (上)	385
6.4.4 接口与类型转换	238	9.1 Delphi 对象的基础: VCL	385
6.5 接口和多重继承	239	9.1.1 VCL 的层次结构	385
6.5.1 什么是多重继承	239	9.1.2 组件的继承关系	387
6.5.2 利用接口实现多重继承	240		
6.5.3 有侧重的多重继承	243		
6.5.4 多重继承的深入讨论	248		
6.6 接口和面向对象编程	252		

9.2 TObject: 所有对象的根	388	10.2 TString、TList、TCollection: 列表与集合	432
9.3 TPersistent: 持久对象	392	10.2.1 TString 与 TStringList	432
9.4 TComponent: 组件对象	396	10.2.2 TList	436
9.4.1 概述	396	10.2.3 TCollection	437
9.4.2 属性	399	10.3 TStream: 流对象与流化存储技术	442
9.4.3 方法	403	10.3.1 TStream 类及其派生类	442
9.4.4 组件的从属关系	406	10.3.2 TFileStream 与 TMemString	445
9.5 TApplication: 应用程序对象	407	10.3.3 TCompressionStream 和 TDecompressionStream	446
9.5.1 概述	407	10.4 VCL 的可视化工作机制	449
9.5.2 属性	408	10.4.1 TFile 类、TReader 类和 TWriter 类	449
9.5.3 方法	412	10.4.2 TStream 和组件属性的存取	451
9.5.4 事件	413	10.4.3 Object Inspector 的工作原理	455
第 10 章 深入浅出 VCL (下)	417	附录 A ModelMaker 使用指南	459
10.1 TThread: 线程对象	417	参考文献	475
10.1.1 概述	417		
10.1.2 线程对象的封装和运行机制	417		
10.1.3 使用线程对象	423		

第 1 章 建立面向对象的新思维

1.1 导论

1.1.1 历史背景

软件开发的过程就是人们使用各种计算机语言将自身关心的现实世界映射到计算机世界的过程。

数字计算机的先驱——第一台加法机，是 1642 年由法国科学家、数学家兼哲学家布莱斯·帕斯卡（Blaise Pascal）设计的。这个装置使用了一系列有 10 个齿的轮子，每个齿代表从 0 到 9 的一个数字。轮子互相连接，从而通过按照正确的齿数向前移动轮子，就可以将数字彼此相加。后来 Pascal 这个伟大的名字被用来为一种广泛使用的计算机语言命名，Pascal 语言经过不断发展，注入了面向对象、可视化、RAD（Rapid Application Development）等最具活力的要素，就变成了现在我们所用的 Delphi。

1. 开端

当代计算机的数学理论基础是由计算机的开山鼻祖——大名鼎鼎的图灵于 1937 年提出的图灵机模型。随后不到 10 年，电子计算机就诞生了（1945）。第一台电子计算机 ENIAC（见图 1-1）含有 18 000 个真空管，具有每分钟几百次的运算速度，但是最初程序是通过导线传送到处理器内的，必须由人工更改。它当时的主要任务之一就是用于导弹弹道轨迹的计算。当时的软件开发（如果可以称之为软件开发的话）与现在的大不相同。为了算一道题，要有人事先把完成加减乘除等各类运算的部件像搭积木那样搭起来，如果换一道题，则要把这些部件分解开来，再根据新的要求重新搭建，与现在相比效率极低。



图 1-1 第一台电子计算机 ENIAC

现代电子计算机的体系结构及实际计算模型来自冯·诺依曼的思想。1946年他和同事们发现了ENIAC的缺陷,发表了一份报告,提出了程序放入内存,顺序执行的思想,这样,当算一道新题时就只需采取改变计算机中程序的“软”的方法。英国的科学家威尔克斯实现了冯·诺依曼的思想,领导研制了“艾克萨克”,“艾克萨克”在技术上比之ENIAC,有了巨大飞跃。因此,现在的计算机通常被称为冯·诺依曼计算机。

软件开发的历史也从此正式开始。

早期的程序员们使用机器语言来进行编程运算。随着编译技术的出现,人们设计了许多更高级别的语言:这些语言摆脱了机器语言繁琐的细节,更接近于人的自然语言,并因此而迅速流行开来。据统计,全世界的高级语言起码有几千种,但从可计算性的角度看,它们的计算能力都等价于图灵机。已经证明,一个计算机语言,只要除了赋值语句之外,还包括顺序语句、条件语句和循环语句,它的计算能力即相当于图灵机。这里当然要排除其他技术因素的影响,如程序长度、变量个数、数据精度等。

由于图灵机的想法是把问题转化为一步一步按规则执行的机械求解过程,各种计算机语言也不过是某种形式语言,因此软件开发的过程实质上就是程序员们对客观世界问题域的形式化的过程。程序员们先建立问题的模型(形式化),再用计算机语言加以合适的表达,最后再输入计算机里进行计算。

针对日趋复杂的软件需求的挑战,软件业界发展出了面向对象(OO)的软件开发模式。目前作为针对“软件危机”的最佳对策,OO技术已经引起人们的普遍关注。OO技术最初被多数人看做只是一种不切实际的方法和满足一时好奇心的研究,但现在却得到了人们近乎狂热的欢迎。许多编程语言都推出了支持面向对象的新版本。大量的面向对象的开发方法被提出来。关于OO的会议、学术研讨班和课程极受欢迎。无数专业的学术期刊都为这一话题开辟了专门的版面。一些软件开发合同甚至也指明了必须使用OO的技术和语言。20世纪90年代的面向对象软件开发技术就像当年20世纪70年代时的结构化软件开发技术一样让人着迷,而且OO的发展势头还在日益加速。

2. 早期面向对象语言及其影响

诸如“对象”和“对象的属性”这样的概念,可以一直追溯到20世纪50年代初。它们首先出现于关于人工智能的早期著作中。然而,OO的实际发展却是始于1966年。当时Kisten Nygaard和Ole-Johan Dahl开发了具有更高级抽象机制的Simula语言。Simula提供了比子程序更高一级的抽象和封装;并且为仿真一个实际问题,引入了数据抽象和类的概念。大约在同一时期,Alan Kay正在犹他大学的一台个人计算机上努力工作,他希望能在其上实现图形化和模拟仿真。尽管由于软硬件的限制,Kay的尝试没有成功,但他的这些想法并没有消失。20世纪70年代初期,他加入了Palo Alto研究中心(PARC),再次将这些想法付诸实施。

在PARC,他所在的研究小组坚信计算机技术是改善人与人、人与机器之间通信渠道的关键。在这个信念的支持下,并吸取了Simula的类的概念,他们开发出了Smalltalk语言;1972年PARC发布了Smalltalk的第一个版本。大约在此时,“面向对象”这一术语正式确定。Smalltalk被认为是第一个真正面向对象的语言。Smalltalk的目标是为了使软件设计能够以尽可能自动化

的单元来进行。在 Smalltalk 中一切都是对象——即某个类的实例。最初的 Smalltalk 世界中，对象与名词紧紧相连。Smalltalk 还支持一个高度交互式的开发环境和原型方法。这一原创性的工作开始并未发表，只是被视为带浓厚试验性质的学术兴趣而已。

20 世纪 80 年代，人们对图形用户界面（GUI）开始感兴趣。Xerox 公司和后来的 Apple 公司创造了现在广泛应用的 WIMP 界面[○]。Smalltalk 的许多思想和研究导致了在窗口（Window）、图标（Icon）、鼠标（Mouse）和指点式（Pointer）GUI 环境开发上的一系列进展。

Smalltalk 语言还影响了 20 世纪 80 年代早期和中期的面向对象的语言，如：Objective-C（1986）、C++（1986）、Self（1987）、Eiffel（1987）、Flavors（1986）。面向对象的应用领域也被进一步拓宽。对象不再仅仅与名词相联系，还包括事件和过程。1980 Grady Booch 首先提出面向对象设计（OOD）的概念，其他人紧随其后，此后，面向对象分析的技术开始公开发表。1985 年，第一个商用面向对象数据库问世。20 世纪 90 年代以来，面向对象的分析、测试、度量和管理等研究都得到了长足发展。目前对象技术的前沿课题包括设计模式（design pattern）、分布式对象系统和基于网络的对象应用等。

3. 面向对象语言

最早的高级语言大约诞生于 1945 年，是德国人楚译为他的 Z-4 计算机设计的 Plan Calcul，比第一台电子计算机还早几个月；在电子计算机上实现的第一个高级语言是美国尤尼法克公司于 1952 年研制成功的 Short Code；而真正得到推广使用，至今仍在流行的第一个高级语言是美国的计算机科学家巴科斯设计，并于 1956 年首先在 IBM 公司的计算机上实现的 FORTRAN 语言。

早期的高级语言主要是应用于科学和工程计算，其代表作有 FORTRAN 和 ALGOL60。计算机进入商业和行政管理领域以后，出现了 COBOL 和 RPG 等便于商界使用的语言。近年来，这类语言和数据库技术、图形界面技术（可视化编程）、面向对象的思想及 RAD 的概念相结合，形成了一批更方便使用的所谓第四代语言（4GL），如 PowerBuilder、Delphi、VB 等。

前一类应用于科学和工程计算的大型语言相对来说更为基础，因而也更为灵活，应用范围也更为广泛。在 FORTRAN、BASIC 之后，自 20 世纪 70 年代以来，结构化特征明显，简单易用、可靠性强的 PASCAL 异军突起，在世界范围内广泛流行。但进入 20 世纪 80 年代以后，它的地位又逐渐为更实用的 C 语言所替代。到现在，C 语言的地位已相当于一种“高级汇编语言”了。

20 世纪 80 年代后期，面向对象的语言开始浮出水面，C++ 借助使用 C 语言的庞大程序员队伍，一举建立了面向对象语言的老大地位。从面向对象的思想正式统治了整个软件开发界。C++ 的流行甚至使得美国军方从 1980 年开始大力扶植的 Ada 语言还未及推广便“胎死腹中”了。

20 世纪 90 年代以后，计算机世界更是发生了天翻地覆的变化，原先的单机平台让位于 Web，“网络就是计算机”，新的语言不仅要是 OO（面向对象）的、Visual（可视化）的，更要

[○] 即：Window、Icon、Mouse/Menu、Pointer 的缩写，它指的是一种利用了这四者的图形用户界面风格。

是基于 Web (万维网) 的。Java 语言借 Internet 的东风, 横空出世, 一夜红遍天下, 变化之快令人瞠目结舌。

目前, 面向对象的语言包含 4 个基本的分支:

- 基于 Smalltalk 的: 包括 Smalltalk 的 5 个版本, 以 Smalltalk-80 为代表。
- 基于 C 的: 包括 Objective-C、C++、Java、C#。
- 基于 LISP 的: 包括 Flavors、XLISP、LOOPS、CLOS。
- 基于 PASCAL 的: 包括 Delphi (Object Pascal)、Turbo Pascal、Eiffel、Ada 95。

Simula 实际上是所有这些语言的老祖宗。在这些 OO 语言中, 术语的命名和支持 OO 的能力都有不同程度的差别。尽管 Smalltalk-80 不支持多继承, 但它仍被认为是最面向对象的语言。

在基于 C 的 OO 语言中, Objective-C 是 Brad Cox 开发的, 它带有一个丰富的类库, 已经被成功用于大型系统的开发。C++ 是由贝尔实验室的 Bjarne Stroustrup 编写的。它将 C 语言中的 STRUCT 扩展为具有数据隐藏功能的 CLASS。多态性通过虚函数 (virtual function) 来实现。C++ 2.0 支持多重继承。在多数软件领域, 尤其是 UNIX 平台上, C++ 都是首选的面向对象编程语言。同 C 和 C++ 相类似的新一代基于 Internet 的面向对象语言 Java 是由 Sun 公司研制的。它于 1995 年伴随着 Internet 的崛起而风靡一时。用 Java 写的 applet 可以嵌入 HTML 中被解释执行, 这使它具备了跨平台特性。Java 和 Ada 一样支持多线程和并发机制, 又像 C 一样简单、便携。

基于 LISP 的语言, 多被用于知识表达和推理的应用中。其中 CLOS (Common LISP Object System) 是面向对象 LISP 的标准版。

在基于 Pascal 的语言中, Delphi 是 Borland 公司在 Object Pascal 基础上开发的。Eiffel 是由交互软件工程公司的 Bertrand Meyer 于 1987 年发布的。它的语法类似 Ada, 运行于 UNIX 环境。Ada 在 1983 年刚出来时并不支持继承和多态性, 因而不是面向对象的。到了 1995 年, 一个面向对象的 Ada 终于问世, 这就是 Ada 95。

除了上述的面向对象的语言之外, 还有一些语言被认为是基于对象 (object-based) 的。它们是: VB、Alphard、CLU、Euclid、Gypsy、Mesa、Modula。

计算机语言的层出不穷, 表面看来是编程工具在不断推陈出新, 但其背后反映的却是一种更为深刻的认识论的改变, 即你是用何种观点来认识这个世界的。

但语言仅仅是一个工具而已, 使用面向对象语言的人, 不一定就是在用面向对象的思想进行编程。君不见有学 C++ 的人, 还在写面向过程的程序吗? 他们只知道循环、分支、判断, 而不知道对象、多态、模式。Ian Graham 说得好: “一个系统或语言是不是面向对象的并不重要, 重要的是怎样才能是面向对象的以及用什么办法实现相关的好处”。(《面向对象方法: 原理与实践》机械工业出版社 2003 年 3 月出版)

1.1.2 面向过程和面向对象

在面向对象编程中, 程序被看做是相互协作的对象集合, 每个对象都是某个类的实例, 所有的类构成一个通过继承关系相联系的层次结构。面向对象的语言常常具有以下特征: 对象生

成功能、消息传递机制、类和遗传机制。这些概念当然可以并且也已经在其他编程语言中单独出现，但只有在面向对象语言中，它们才共同出现，并以一种独特的合作方式互相协作、互相补充。

面向过程编程模式如图 1-2 所示。在这种编程模式中，数据和函数是分开的，即程序员看到的是函数或过程的集合以及单独的一批数据。程序的处理过程如下：

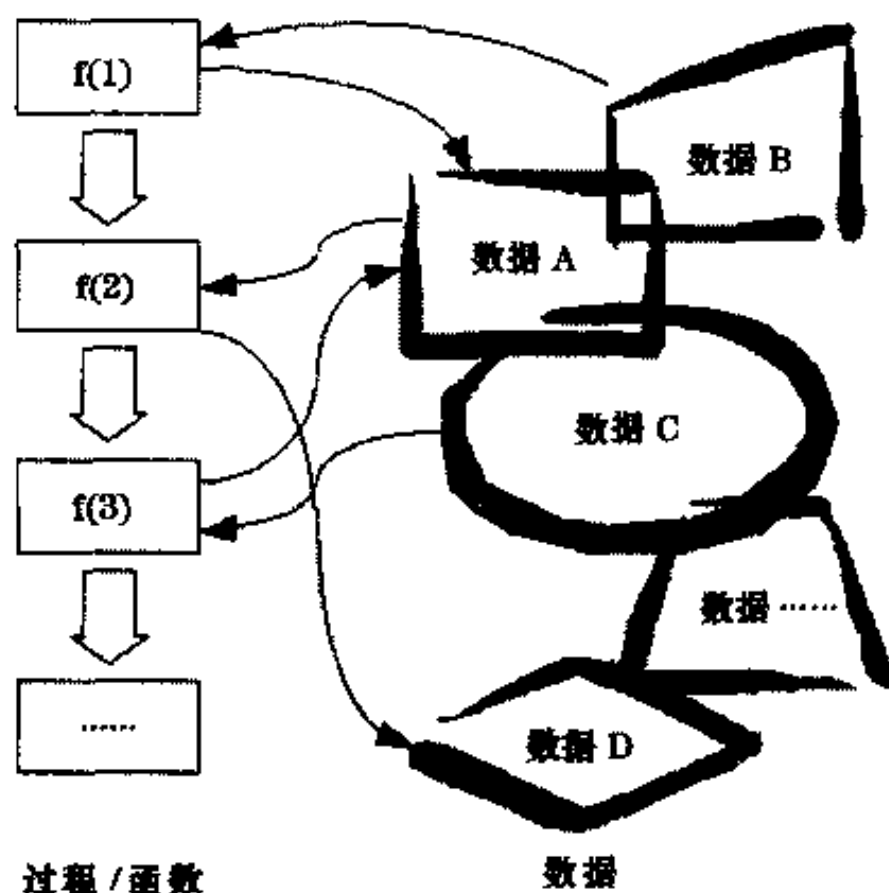


图 1-2 过程化编程模式

参数输入→|函数/过程代码|→结果输出

为实现某个功能，参数被传入某个处理过程，最后传回计算结果。比如数据 B 传入函数 f (1)，产生结果数据 A，数据 A 再传入 f (2)。如此重叠的数据存取使得并行性和数据完整性问题变得相当复杂。对于程序维护人员来说，无论是函数还是数据结构的改动，都会使整个程序受到干扰。因此，程序的维护和扩展几乎难以进行。

面向对象编程模式如图 1-3 所示。在这种模式中，函数和它需要存取的数据封装在称为对象的包中。对象之间的数据访问是间接的，是通过接口进行的。我们可以将对象看做是鸡蛋，蛋黄是数据；蛋清是访问数据的函数；蛋壳代表接口（即那些公开或公布的方法和属性）。蛋壳接口隐匿了函数和数据结构的实现。当数据结构和内部函数变化时，这种变化被限制在内部的局部范围内。由于接口的相对稳定性，使得这种内部变化的影响不会波及到其他对象，除非蛋壳破裂（接口发生变化）。因而面向对象模式开发的程序是易于维护和扩展的。

在面向对象的编程模式中，程序的功能是通过与对象的通信获得的。对象是被定义为一个封装了状态（数据）和行为（操作）的实体。

状态实际上包含了执行行为的信息，它以数据形式存于对象之中。消息是对象通信的方式，因而也是获得功能的方式。对象收到发给它的消息后，或者执行一个内部操作（有时成为方法或过程），或者再去调用其他对象的操作。

实际上，软件开发的过程就是人们使用各种计算机语言将自身关心的现实世界（问题域）

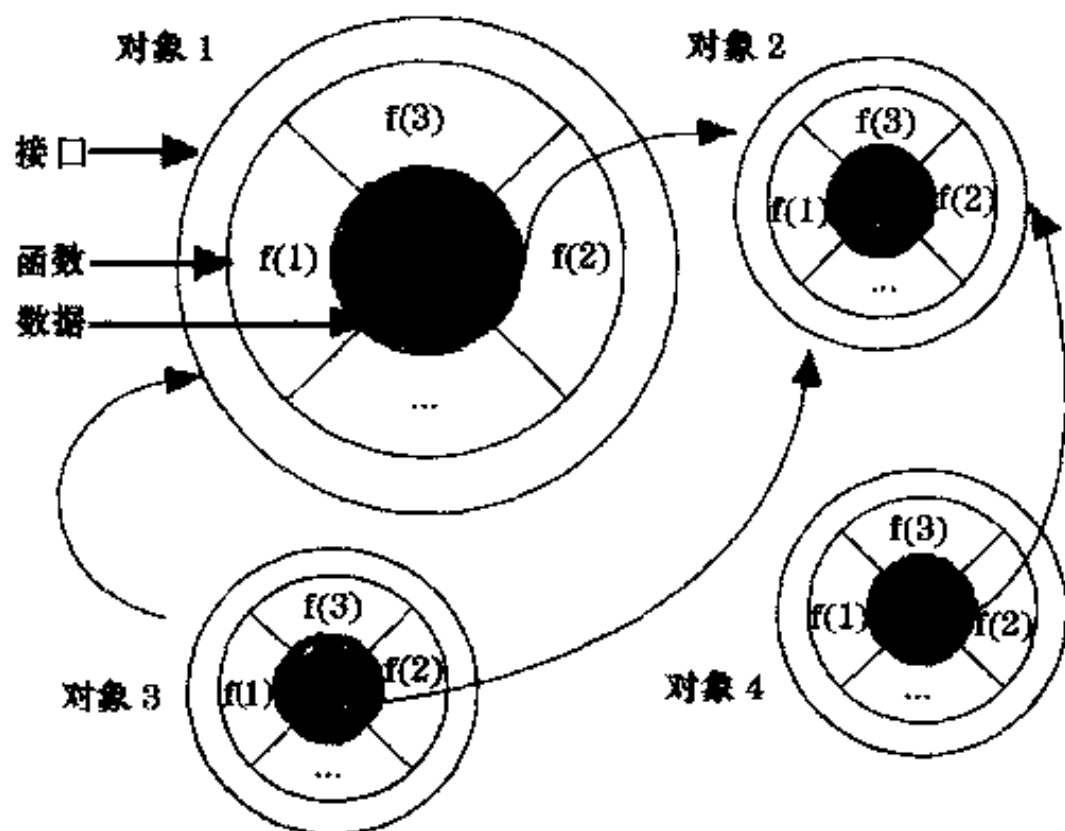


图 1-3 面向对象编程模式

映射到计算机世界的过程，这个过程通常是这样的：

现实世界问题域 → 建立模型（借助某种建模思想） → 编程实现（借助某种编程语言） → 计算机世界执行求解

通过前面的分析，我们可以看出，在面向过程编程模式中，程序员们分析了问题域之后，得到一个面向过程的模型，其中常见的词语是变量、函数、过程等；它的过程是：

现实世界 → 面向过程建模（流程图、变量、函数） → 面向过程语言 → 执行求解

在面向对象编程模式中，程序员们先得到一个面向对象的模型，其中常见的词语是类、对象、方法、消息等；它的过程是：

现实世界 → 面向对象建模（类图、对象、方法） → 面向对象语言 → 执行求解

由此可见，面向对象更接近于计算机世界的物理实现；面向对象思想则更符合于人们的认识习惯。如果说，软件危机的根源在于人们认识表达的过程（不断反复，逐步深化）和冯·诺依曼计算机的实现过程（顺序执行）之间存在巨大鸿沟的话，那么从面向过程到面向对象则意味着这鸿沟在逐渐缩小；面向对象作为一种思想及编程语言，为软件开发的整个过程——从分析设计到编码——提供了一个完整解决方案。

1.1.3 面向对象的技术背景和特点

面向对象（OO）是一种方法，一种思想，同时又是一种技术。它力求更客观自然地描述现实世界，使分析、设计和实现系统的方法同认识客观世界的过程尽可能一致。

“对象”（Object）一词，早在19世纪就由现象学大师胡塞尔提出并定义。对象是客观世界中的事物在人脑中的映像，这种映像通过对同一类对象的抽象反映成人的意识，并做为一种概念而存在。例如：当我们认识到一种新的物体，它叫苹果，于是在我们的意识当中就形成了苹

果的概念。这个概念会一直存在于我们的思维当中，并不会因为这个苹果被吃掉而消失。这个概念就是现实世界当中的物体在我们意识当中的抽象。只要这个对象存在于我们的思维意识当中，我们就可以藉此判断同类的东西。譬如，当我们看到另外的苹果时，并不会因为所见的第一个苹果不在了失去了供参照的模板而不认识苹果了。当我们接触某些新事物时，我们的意识就会为这些事物确立一个对象，并把相似的对象抽象成一个类的概念。

客观世界是由许多不同种类的对象构成的，每一个对象都有自己的运动规律和内部状态，不同对象之间相互联系、相互作用。面向对象技术是一种从组织结构上模拟客观世界的方法，它从组成客观世界的对象着眼，通过抽象，将对象映射到计算机系统中，又通过模拟对象之间的相互作用、相互联系来模拟现实客观世界，描述客观世界的运动规律。

传统的面向过程的功能分解法属于结构化分析方法，分析者将对象系统的现实世界看做为一个大的处理过程，然后将其分解为若干个子处理过程，直至将整个系统分解为各个易于处理的过程为止，然后解决系统的总体控制问题。在分析过程中，用数据描述各子处理过程之间的联系，整理各个子处理过程的执行顺序。这种方法缺乏对问题的基本组成对象的分析，不够完备，尤其是当需求功能变化时，将导致大量修改，不易维护。

面向对象技术以基本对象模型为单位，将对象内部处理细节封装在模型内部，并且重视对象模块间的接口联系和对象与外部环境间的联系，能层次清晰地表示对象模型。

面向对象方法则从根本上对问题域中的对象及其关系进行详尽的分析，并在此基础上完成需求功能，力求使对系统的修改和增加功能变得很容易，修改时不至于对系统结构产生大的影响。

面向对象技术有着自身鲜明的特点。首先要搞清什么是对象。客观世界中对象是形形色色的，常可以划分成不同类，不同类的对象又是千差万别的。例如自然界中的对象是看得见摸得着的各类实体，而各类生产活动中的对象则是处理或控制过程，程序设计中的对象却是数据结构等。把所有这些概括为对象，不难看出它们有以下几个共同特点：

- 某类对象是对现实世界具有共同特性的某类事物的抽象。
- 对象蕴含着许多信息，可以用一组属性来表征。
- 对象内部含有数据和对数据的操作。
- 对象之间是相互关联和相互作用的。

面向对象技术，正是利用对现实世界中对象的抽象和对象之间相互关联和相互作用的描述来对现实世界进行模拟，并且使其映射到目标系统中的。所以，面向对象的特点主要概括为抽象性、继承性、封装性和多态性。

- 抽象性——指对现实世界中某一类实体或事件进行抽象，从中提取共同信息，找出共同规律，反过来又把它们集中在一个集合中，定义为所设计目标系统中的对象。
- 继承性——新的对象类由继承原有对象类的某些特性或全部特性而产生出来，原有对象类称为基类（或称超类），新的对象类称为派生类（或子类，在本书中统一称为派生类），派生类可以直接继承基类的共性，又允许派生类发展自己的个性。继承性简化了对新的对象类的设计。

- 封装性——是指对象的使用者通过预先定义的接口关联到某一对象的服务和数据时，无需知道这些服务是如何实现的。即用户使用对象时无需知道对象内部的运行细节。这样，以前所开发的系统中已使用的对象能够在新系统中重新采用，这就减少了新系统中分析、设计和编程的工作量。
- 多态性——是指不同类型的对象可以对相同的激励做出适当的不同响应的能力。多态性丰富了对象的内容，扩大了对象的适应性，改变了对象单一继承的关系。

1.1.4 为什么要使用面向对象的编程技术

对象的概念对软件解决方案具有莫大的好处，在设计优秀、合理的情况下尤其如此。你可以只编写一次代码而在今后反复重用，而在非 OOP（面向对象编程）的情况下则多半要在应用程序内部各个部分反复多次编写同样的功能代码。所以说，由于面向对象编程减少了编写代码的总量，从而加快了开发的进度，同时降低了软件中的错误量。

用来创建对象的代码还可能用于多个应用程序。比方说，你的团队可以编写一组标准类来计算你的可用资源，然后用这些代码在所有需要同类对象的解决方案中创建对象，比如客户定单接口、股票价值报表和发给销售队伍的通知等等。

OOP 的另一优点是对代码结构的影响。像继承之类的面向对象概念通过简化变量和函数的方式而便利了软件的开发过程。OOP 可以更容易地在团队之间划分编码任务。同时，由于采用 OOP，辨别派生类代码的依附关系也变得更简单了（比如说继承对象的代码）。此外，软件的测试和调试也得以大大简化。

但是 OOP 也存在一些固有的缺点。假如某个类被修改了，那么所有依赖该类的代码都必须重新测试，而且还可能需要重新修改以支持类的变更。此外，如果文档没有得到仔细的维护，那么我们很难确定哪些代码采用了基类（被继承的代码）。假如在开发后期发现了软件中的错误，那么它可能影响应用程序中相当一部分的代码。

面向对象编程在编程思想上同传统开发不同，它需要开发人员转变传统开发中所具备的惯性思维方式。对一个有经验的 OOP 开发队伍来说，采用 OOP 的好处是显而易见的。如果你正在考虑转向 OOP，那么就必须保证已经拥有了富有经验的主要开发人员能负责地检查软件中的缺陷和体系结构。

下面我们就看看 OOP 技术到底能做些什么。

对象是建立面向对象程序所依赖的基本单元。用更专业的话来说，所谓对象就是一种代码的实例，这种代码执行特定的功能，具有自包含或者封装的性质。这种封装代码通常叫做类、对象类或者模块或者在不同编程语言中所应用的其他名称。以上这些术语在含义上稍微有些不同，但它们都是代码的集合。

正如上而提到的那样，对象本身是类或者其他数据结构的实例。这就是说，现有的物理代码起到了创建对象的模板作用。执行特定功能的代码只需要编写一次却可被引用多次。每一种对象具有自己的标识，也就是令对象相互区别的对象名称。

对象并不是类的实际拷贝。每一对象都有自己的名称空间，在这种名称空间中保存自己的

标识符和变量，但是对象要引用执行函数的原有代码。

“封装”的对象具有自己的函数，这种函数被称做“方法”，而对象的变量则被称为属性。当对象内部定义了属性的时候，它们通常不能扩展到实例以外。假设现有一个类叫 vegetable (蔬菜)，同时又创建了两个对象实例 carrot (胡萝卜) 和 celery (芹菜)，那么我给 carrot 设置的值就不会影响到 celery 内部的值。vegetable 自身内部的变量却永远不会得到定义，因为 vegetable 类只是一种模板。

在特定的场合下，有些函数确实会影响类而不是由类所创建的对象。类属性指的是专门设计来保留对象之间所用的值。类方法则用来定义和跟踪类属性。

某些编程语言可以让用户调用类的函数而不是创建整个实例。如果函数被分配以标识符 (或者句柄)，在某些情况下它们就可以被视做具有自身权限的对象。不过，在大多数情况下函数只是用来实现某种结果的方法。

在主程序里，定义对象的类通过实例化的方式构造对象。对象所具有的所有方法都可以用来创建所希望的结果，而属性则可以被引用和操作。当不再需要对象时，主程序可以清除对象。换言之，对象是有生死的，有生命期的。

对象类有一种功能强大的特性，这就是它们可以继承其他类。这就意味着，如果我们编写了某个 potato (土豆) 类，那么它就可以继承 vegetable 类而防止我们重新编写已经存在的功能。vegetable 类可用的所有函数都可以被 potato 类使用。进而，vegetable 又可以继承 food (食品) 类，以此类推。

某些 OOP 编程语言还具有多重继承的概念。比如说，potato 类可以继承 vegetable 和 starch (淀粉) 类。不过这样可能会产生一些问题，比如两种类都具有同样名称的一些属性。在具体处理多重继承概念的时候各种语言的方式是不同的，某些语言完全禁用这一概念。Delphi 虽然禁用多重继承，但仍然可以通过接口技术来实现类似的功能。

在继承了类后，我们可以通过覆盖 (override) 方法来获得希望的结果。比如，我的 vegetable 类可能有一个函数名叫 prepare，该方法主要指导你如何备菜。可是，在实例化 potato 类的时候我希望其中包含与土豆有关的特殊定义，于是可以创建一个函数，它的名字和蔬菜类中的备菜函数名一样但却修改了原有的函数行为。如果没有覆盖 prepare 方法，则用到的是 vegetable 类中的函数。这就叫多态性 (polymorphism)。

多态性的另一方面涉及到对象方法的类型一致性问题。这样有助于保证所引用的函数具有以下关系：如果能够实例化 vegetable 对象，那么就应该能够实例化 potato 对象。这是因为 potato 是 vegetable 的派生类。可是，因为 vegetable 并不是 potato 的派生类，所以反过来的实例化却是不允许的。如果实例化了 potato 对象，那么就不再需要实例化 vegetable 对象了。

如何定义多态性有各种观点，而其最终用途却是同样的。无论如何，这是一种重要的 OOP 概念。再结合继承技术，显然，OOP 为什么具有如此强大开发功能的原因不言自明。

面向对象的开发强调从问题域的概念到软件程序和界面的直接映射；心理学的研究也表明，把客观世界看成是许多对象更接近人类的自然思维方式。对象比函数更为稳定；软件需求的变动往往是功能相关的变动，而其功能的执行者——对象——通常不会有大的变动。另外，

面向对象的开发也支持、鼓励软件工程实践中的信息隐藏、数据抽象和封装。在一个对象内部的修改被局部隔离。面向对象开发的软件易于修改、扩充和维护。

面向对象也被扩充应用于软件生命周期的各个阶段——从分析到编码。而且，面向对象的方法自然而然地支持快速原型法和 RAD（快速应用开发）。面向对象开发的使用鼓励重用，不仅包括软件的重用，还包括分析、设计模型的重用。更进一步，OO 技术还方便了软件的互换性，即网络中一个节点上的应用能够利用另一个节点上的资源。面向对象的开发还支持并发、层次和复合等一些在目前的软件系统中常见的要求。今天我们常常需要建造一些软件系统，而不仅仅是一个黑盒应用。这些复杂系统通常包含由多个子系统组成的层次结构。面向对象的开发支持开放系统的建设；利用不同的应用来进行软件集成有了更大的柔性。最后，面向对象开发的使用可以减小开发复杂系统所面临的危险，主要是因为系统集成遍布软件生命周期的各个阶段。

1.2 面向对象的基本概念

OOP 就是使用对象进行编程的过程，所谓对象就是协调数据存储以及作用于数据之上操作的独立实体。对象把数据保存在属性（变量、域、数据成员）中。对象中也包括作用于属性之上的操作，称之为方法（函数、过程、子程）。比如，设想把一辆汽车作为一个对象。一些信息或数据描述了许多汽车中的这一辆的品牌、颜色、最高时速，这些都是汽车对象的属性。汽车也可以完成诸如：启动、加速、行驶、绕行、减速和停车等操作。这些是汽车对象的操作，用 Delphi 面向对象语言的术语来说，就是它的方法。

用户可以通过定义一个对象集合以及它们之间的相互作用来创建一个面向对象程序。许多对象协同工作来定义一个完成用户需要的程序。在本书后面将对面向对象设计过程进行深层次的剖析。在这里，只打算让用户先熟悉面向对象的一些概念，并加深用户对 Delphi 面向对象编程的理解。

1.2.1 类和对象

那么当用户创建一个面向对象程序时，是如何建立对象的呢？可以通过类声明来定义类，然后使用类来创建用户需要的对象。类声明是用来创建对象的模板的抽象规格说明。当用户编写自己的 Delphi 程序时，所涉及到的主要工作就是编写类声明。当程序运行时，已声明的类用来创建新对象。由类创建对象的过程称为实例化（instantiation）。每个对象是类的一个新实例（instance）。

图 1-4 显示了类和对象的不同之处。汽车类是对什么是汽车的一个定义，而解放、红旗和奔驰是对象，是汽车类的实例。

1. 属性

类定义中的属性指定了使一个对象区别于其他对象的值。比如，在汽车类的定义中包括汽车的品牌、颜色、最高时速这些属性，如图 1-5 所示。每个对象的这些属性都有自己的值。所有的由类定义建立的对象都共享类的方法。但是，它们都拥有在类方法中定义的所有变量的副本（copy）。

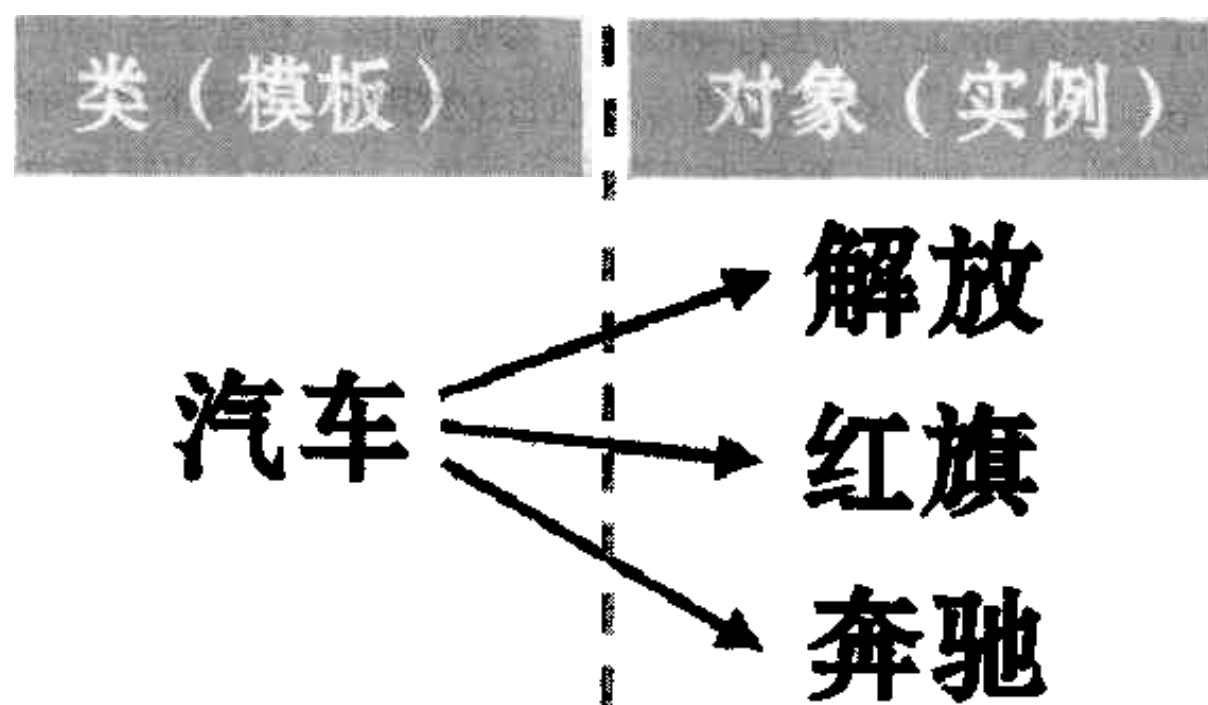


图 1-4 类和类对象

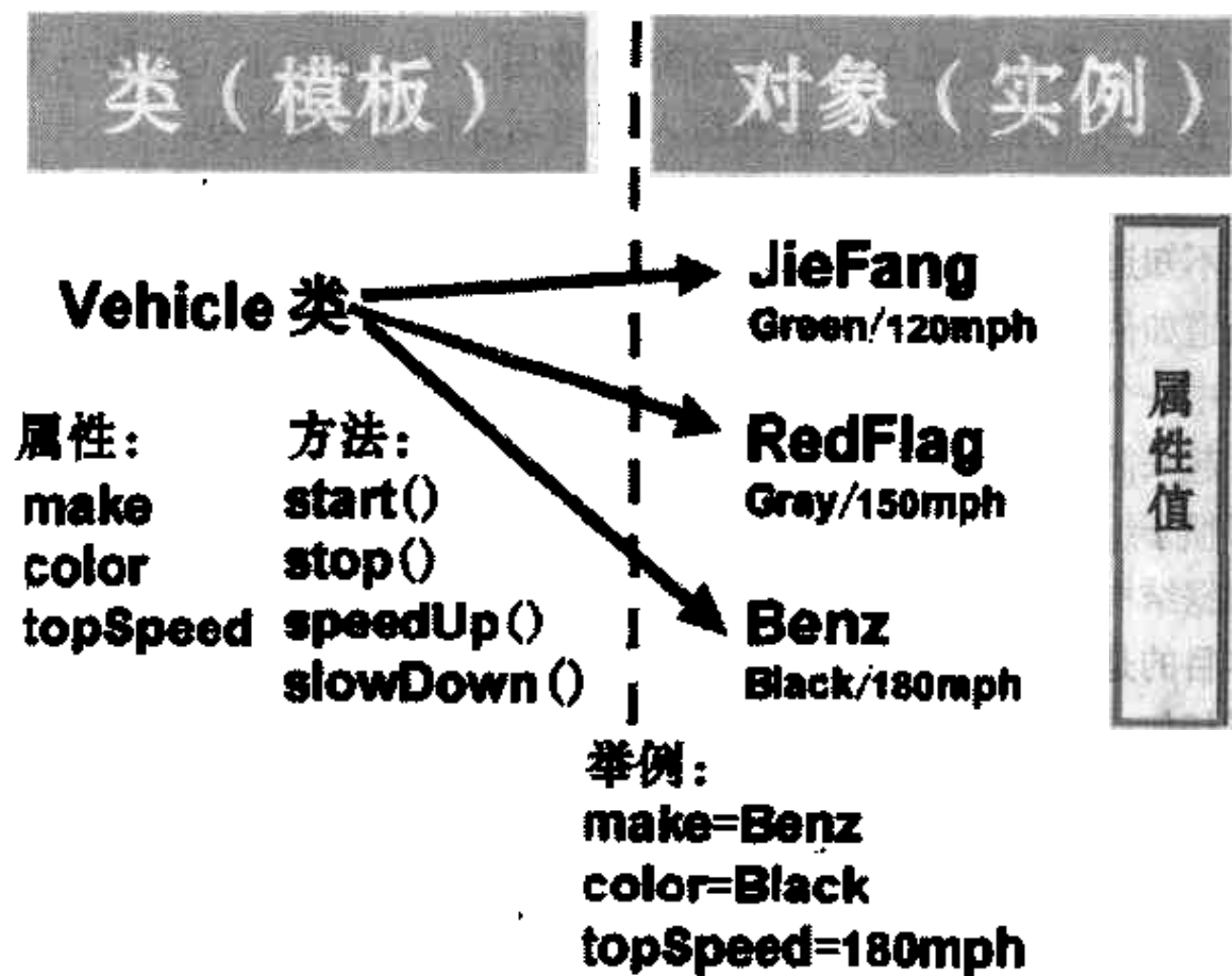


图 1-5 拥有属性和方法的类和对象

2. 方法

对象的操作由 Delphi 的方法来指定。要使一个对象做某件事情，就要调用它的相应方法。在用户程序中，这由一行给出了方法名及参数列表的代码来完成。假如用户想要改变汽车的颜色。在程序中，可以编写如下代码：

```
mycar. setColor (yellow);
```

这行代码包括了对象名 `myCar`；方法名 `setColor`；参数 `yellow`（包含于括号中）。可以看到，一个方法的参数是传送给方法的数值，这些数值在方法执行中被用到。在这个例子中，`setColor` 方法把数值 `yellow` 赋给汽车对象的 `color` 属性，然后改变汽车的颜色。这个过程也被称为向对象发送一条消息。在本例中，用户向汽车对象发送了一条消息，要求改变颜色属性，并指定了新的颜色值。

有时为了区分到底是一个对象的属性还是方法，就在方法名后加上括号，比如 `stop()`。当参数传递给方法时，它们在括号中给出。

方法也可以返回一个数据值。因而下面这个方法：

```
myCar.getColor();
```

将返回汽车的颜色值，在本例中为黄色。

1.2.2 封装

封装（encapsulation）是一个面向对象的术语。它的意思很简单，就是把东西包装起来。换言之，属性定义和方法都包装于类定义之中，类定义可以看成是封装构成类的属性和方法。通过限定类成员的可见性，可以使得类成员中的某些属性和方法能够不被程序的其他部分看见，它们“隐藏”了起来，它们只能在所定义的类中使用。从表面看来，这是一个非常简单的概念，但却在经历了30年的程序设计实践和深入思考后才得到一致认可。

在封装出现以前，程序的任何部分都能存取其他的任何部分，这使得改动变得十分困难，因为程序员永远不知道这种改动会产生什么样的影响。当发现一个变量拥有一个糟糕的值时，用户永远也不知道如何变成了这样的情况。封装简化了编程和维护，因为程序改动的副作用往往被局限于程序中一个小的区域内，这极大地缩小了潜在问题的影响范围。所以，封装的确是个好主意。它看起来简单，但却有如此妙用。

如果用户习惯于组合不同的数据类来创建复合数据结构，那么可以把类定义看做是建立在类合成之上的数据结构，因为它封装的不仅是数据定义，而且是操纵这些数据的方法。

封装的根本目的是保证对象的属性只能通过对象的方法进行存取。这种实现需要额外的编码，但它保证了任何使用该对象的编码都独立于该对象执行的实现细节。这使得用户可以按个人意愿改变对象的执行过程。牢记这样一点：只要对象的程序设计接口不变，也就是对象方法的结构不变，那么任何使用该对象的代码都能像以前一样正常地工作。

1.2.3 继承

图1-6显示了面向对象编程的另一重要方面——继承性。注意类 `Vehicle` 定义了属性 `make`、`color` 和 `topSpeed`。当定义派生类 `Car` 时，它自动继承基类 `Vehicle` 中的属性和方法。类 `Car` 称为一个派生类或子类。类 `Vehicle` 称为 `Car` 的基类。

继承节省了在定义新类中的大量工作，因为编程者可以方便地重用代码。比如，当创建派生类时，`Car.color` 和 `Car.topSpeed` 属性被自动地定义，引用方法 `Car.start()` 时会自动调用在类 `Vehicle` 中定义的方法 `start()`。

但一个派生类不必非得使用继承下来的属性和方法。一个派生类可以选择覆盖（override）

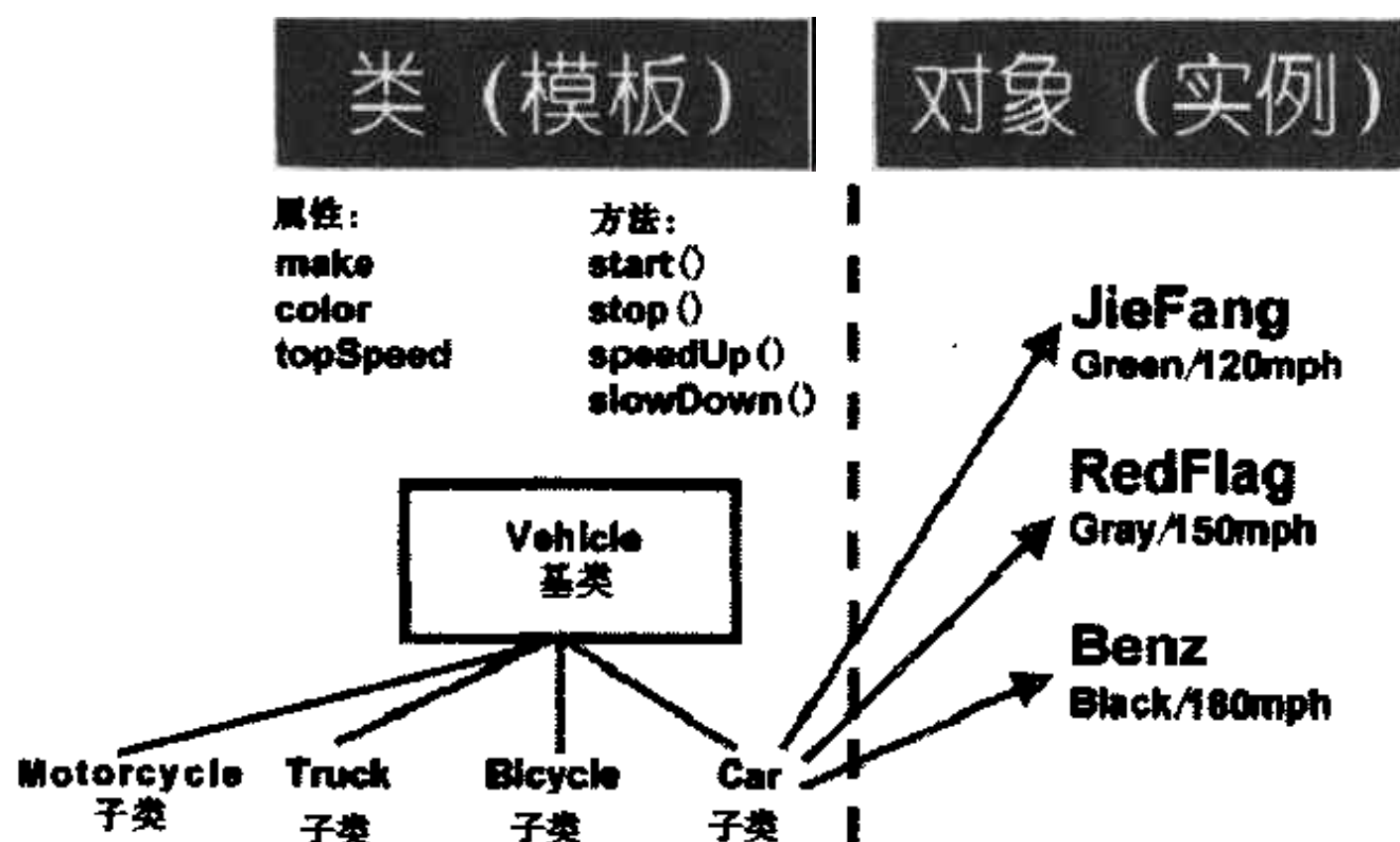


图 1-6 Vehicle 类是 Truck、Motorcycle、Bicycle 和 Car 的基类

已有的属性和方法，或添加新的属性和方法。用户也可添加其他的属性和方法来满足具体的需要——比如说摩托的撑脚架。

只有当用户想向自己新类的定义中添加新的操作，或者把已存在类的缺省行为融合进自己的新类中时，才需要继承一个已存在的类定义。在 Delphi 中，这称为派生一个类。要添加新的方法和属性，只需定义它们即可。要覆盖一个已存在的方法，就要在派生类中（Car）定义一个新方法，该方法与基类中（Vehicle）被覆盖的方法有相同的名字和参数列表。

如果派生类没有覆盖 stop () 方法，那么基类 Vehicle 中的缺省 stop () 方法就被调用。该方法可能关掉发动机。

1.2.4 多态性

这种同样的方法名因为调用的方式不同而执行完全不同的例程的能力带来了面向对象编程中另一个强大的方面——多态性 (polymorphism)。“poly”的意思是“许多”，“morph”意为“形状”。所以，“polymorphism”从词义上说就是许多形态的意义。它在实际中的含义就是不同的对象有相同的一般轮廓或形态，但具体执行的过程却大相径庭。

多态性使用户可以写更加通用的过程。用户可以编写一个过程来控制 Vehicle 对象，说“当看到一个停车标志时，执行 stop () 方法”，这样的通用 stop () 方法与驾驶什么车无关。用户可以让派生类去操心如何完成 stop () 方法，而继续在更高的抽象级别上编写自己的通用过程。比如说，即使 Car 对象的 stop () 方法与 Truck 对象的 stop () 方法完全不一样（卡车在 18 个档之间来回切换），用户也可以编写 Vehicle.stop ()，并知道哪一个合适的方法将被调用。

多态性也使得用户在以后不费多大力气就可以派生程序。假设用户在为小汽车和卡车构建应用程序。用户知道还存在摩托车和自行车，但在当时并不太在意它们。用户写了一个处理 Vehicle 类的程序，把 Truck 和 Car 派生类定义为 Vehicle 的特例。以后，如果需要，无需费劲就

可派生。通过 Vehicle 类来定义 Motorcycle 和 Bicycle 类，并且为 Vehicle 类编写的所有代码都可以为这些新类工作。

方法重载 (overload) 也可以提供类似于多态性的好处。方法重载意味着两个方法有相同的名字但参数不同。方法的名字与它参数的个数及类型决定了该方法的特征 (signature)。一个类可以拥有多个同名的方法，只要每个方法都有不同的特征即可。

注意 返回值的数据类型不参与决定方法的特征。

与多态性一样，方法重载赋予了用户某种程度的概括能力。使用不同的方法特征使用户可以指定同名但细节不同的多种操作。比如，Car 类拥有一个标准的 stop () 方法，即脚踏刹车闸。它也可以有一定义为“紧急刹车”，即双脚猛踩刹车闸的 stop (emergency) 方法，和定义为“冰面刹车”，即轻微地踩刹车闸但有一滑动就松开刹车闸的 stop (ice) 方法。使用相同的名字和不同的参数来调用不同的方法，使用户不必为同一类操作的不同变体而绞尽脑汁取新的名字。这也使类的使用者可以更容易地记住方法的名称。

注意 重载和多态的机制有本质的区别。重载不同于覆盖，准确地讲，它不是面向对象专有的 (详见 5.2 节)。

1.3 面向对象建模和 UML

1.3.1 面向对象建模

在软件开发以程序编写为主的年代，是不会有对建模语言的需求的。美国历次大规模爆发的软件危机让人们软件有了新认识，并且发现，在处理大型、复杂的软件需求时，需要应用一些方法来指导对需求的理解工作；团队化的软件开发方式让人们更加认识到了在团队中的协作和基于标准进行交互的重要性，这些都成为推动建模方法发展的原始推动力。从当前较为流行的建模方法来看，可以大致分为面向过程和面向对象两种。

面向过程的建模技术始于结构化的分析设计技术 (SADT)，而美国空军部 (Air Force) 在 20 世纪 70 年代的集成化计算机辅助制造系统项目中，为了解决制造业人士与 IT 技术人员就需求无法准确、顺利地沟通的问题，在项目的早期定义了 IDEF 系列的图形化建模语言，这套语言很快得到了广泛的认可，并由 IEEE 进行维护与发布。如今，IDEF 成为了面向过程建模技术的代表。

从面向过程建模技术的发展过程中，我们可以清楚地看到：建模语言是一种图形化的文档描述性语言，利用它所期望解决的核心问题就是，沟通障碍的问题，特定行业的专业人士与 IT 行业的专业人士在沟通上的最大障碍就是行业术语，而要解决这个问题，最有效的方式就是要寻找到一种公共语言来进行交互，这是建模语言所要起到的最为重要的作用；同时，建模语言必须支持的是一种思考复杂问题所必须采用的思维方式 (抽象、分解、求精)，并且提供特定问题的特定描述方式，利用不同的图形从各个方面对系统的全部或部分进行描述。

面向对象的建模技术起源于 20 世纪 60 年代的 OOA/OOD，在多年的发展历程中，一直处于一种百家争鸣的发展状态。



比较成熟的面向对象建模技术是 James Rumbaugh 在 1983 年《Object-Oriented Modeling and Design》中谈到的 OMT 的方法。OMT 是一种通过模型来思考问题的方法，这些模型都是围绕着真实世界的概念建立的。OMT 方法提供了一组面向对象的概念及图形符号，然后利用这些概念及符号来分析需求、设计系统、实现，它适应于整个软件的开发过程。OMT 方法是一种思考问题的方法，而不仅仅是一种编程技术。

20 世纪 90 年代初，OMT 和其他面向对象建模技术统一发展成今天广泛使用的 UML（统一建模语言），确定了面向对象建模技术在软件开发中的重要地位。

面向对象的建模是一种新的思维方式，是一种关于计算和信息结构化的新思维。面向对象的建模，把系统看做是相互协作的对象，这些对象是结构和行为的封装，都属于某个类，那些类具有某种层次化的结构。系统的所有功能通过对象之间相互发送消息来获得。面向对象的建模可以视为是一个包含以下元素的概念框架：抽象、封装、模块化、层次、分类、并行、稳定、可重用和可扩展性。

面向对象建模技术的出现并不能算是一场软件革命。更恰当地讲，它是面向过程和严格数据驱动的软件开发方法的渐进演变结果。软件开发的新方法受到来自两个方面的推动：编程语言的发展和日趋复杂的问题域的需求驱动。尽管在实际中，分析和设计在编程阶段之前进行，但从发展历史看却是编程语言的革新带来设计和分析技术的改变。同样，语言的演变也是对计算机体系的增强和需求的日益复杂的自然响应。

为了适应软件开发潮流，目前许多大软件公司纷纷将编程语言和建模工具进行捆绑，比如：Borland 将 Delphi 和 ModelMaker、JBuilder 和 Together 捆绑，微软将 Visual Studio .NET 和 Visio 捆绑，IBM 购买了 Rose。其目的都是在提供一个从建模设计到编码实现的软件应用生命周期解决方案。

虽然影响面向对象技术发展的因素很多，面向对象建模本身也未成熟；但 UML 的出现，使得如何进行面向对象的分析、设计逐渐形成共识，并提供了统一的符号来描述这些活动。面向对象建模领域正在走向统一。面向对象建模已经在以下领域被证明是成功的：空中交通管理、银行、商业数据处理、命令和控制系统、CAD、CIM、数据库、专家系统、图像识别、操作系统、过程控制、空间站软件、机器人、远程通信、界面设计。毫无疑问，面向对象建模的应用将促进软件工业的发展。

1.3.2 UML 是什么

UML 是统一建模语言（Unified Modeling Language）的英文缩写，UML 是一个通用的可视化建模语言，用于对软件进行描述、可视化处理、构造和建立软件系统制品的文档。它可以把人们对所需要构建系统的想法和理解记录下来，以便用于对系统的了解、设计、浏览、配置和维护。UML 适用于各种软件开发方法、软件生命周期的各个阶段、各种应用领域以及各种开发工具，是一种总结了以往建模技术的经验并吸收当今优秀成果的标准建模方法。UML 包括概念的语义、表示法和说明，提供了静态、动态、系统环境及组织结构的模型。它得到了许多可视化建模工具的支持，如：Rose、ModelMaker 等。这些工具提供了代码生成器和报表生成器。UML 标准并没有定义一种标准的开发过程，但它适用于迭代式的开发过程。它是为支持大部分现存

的面向对象开发过程而设计的。

UML 描述了一个系统的静态结构和动态行为。UML 将系统描述为一些离散的相互作用的对象并最终为外部用户提供一定功能的模型结构。静态结构定义了系统中重要对象的属性和操作以及这些对象之间的相互关系。动态行为定义了对对象的时间特性和对象为完成目标而相互进行通信的机制。从不同但相互联系的角度对系统所建立的模型可用于不同的目的。

UML 还包括可将模型分解成包的结构组件, 以便于软件小组将大的系统分解成易于处理的块结构, 并理解和控制各个包之间的依赖关系, 在复杂的开发环境中管理模型单元。它还包括用于显示系统实现和组织运行的组件。

UML 不是一门程序设计语言, 但可以使用代码生成器工具将 UML 模型转换为多种程序设计语言代码, 或使用反向生成工具将程序源代码转换为 UML。UML 是一种通用建模语言, 但是是一种离散的建模语言, 不适合对诸如工程和物理学领域中的连续系统建模。它是一个综合的通用建模语言, 适合对诸如由计算机软件、固件或数字逻辑构成的离散系统建模。

UML 的最终目标是在尽可能简单的同时能够对实际需要建立的系统的各个方面建模。UML 需要有足够的表达能力, 以便可以处理现代软件系统中出现的所有概念, 例如并发和分布, 以及软件工程中使用的技巧, 如封装和组件。它正在成为一种通用语言 (像任何一种通用程序设计语言一样)。UML 提供了多种模型, 不是在一天之内就能够掌握的。它比先前的建模语言更复杂, 因为它更全面。但是不必一下就完全学会它, 而应循序渐进地掌握; 就像学习任何一种程序设计语言、操作系统或是复杂的应用软件一样。

UML 的概念和模型可以分成以下几个概念域:

- **静态结构** 任何一个精确的模型必须首先定义所涉及的范围, 即确定有关应用、内部特性及其相互关系的关键概念。UML 的静态组件称为静态视图。静态视图用类构造模型来表达应用, 每个类由一组包含信息和实现行为的离散对象组成。对象包含的信息被作为属性, 它们执行的行为被作为操作。多个类通过泛化处理可以具有一些共同的结构。子类在继承它们共同的父类的结构和行为的基础上增加了新的结构和行为。对象与其他对象在运行时也具有相互联系的关系。这种对象与对象之间的关系被称为类间的关联。一些元素通过依赖关系组织在一起, 这些依赖关系包括在抽象级上进行模型转换、模板参数的捆绑、授予许可以及通过一种元素使用另一种元素等。另一类关系包括用例和数据流的合并。静态视图主要使用类图。静态视图可用于生成程序中用到的大多数数据结构声明。在 UML 视图中还要用到其他类型的元素, 比如接口、数据类型、用例和信号等, 这些元素统称为类元, 它们的行为很像在每种类元上具有一定限制的类。
- **动态行为** 有两种方式对行为建模。一种是根据一个对象与外界发生关系的生命历史; 另一种是一系列相关对象之间, 当它们相互作用实现行为时的通信方式。孤立对象的视图是状态机——当对象基于当前状态对事件产生反应, 执行作为反应的一部分的动作, 并从一种状态转换到另一种状态时的视图。状态模型用状态图来描述。协作和互操作序列图和协作图来描述。对所有行为视图起指导作用的是一组用例, 每一个用例描述了一个用例参与者或系统外部用户可见的一个功能。

- **实现构造** UML 模型既可用于逻辑分析又可用于物理实现。某些组件代表了实现项目。组件是系统中物理上的可替换的部分，它按照一组接口来设计并实现。它可以方便地被一个具有同样规格说明的组件替换。节点是一种在运行时完成运算任务的资源，该资源定义了一个位置，还包括组件和对象。部署图描述了在一个实际运行的系统中，节点上的资源配置和组件的排列以及组件包括的对象，并包括节点间内容的可能迁移。
- **模型组织** 计算机能够处理大型的单调的模型，但人力不行。对于一个大型系统，建模信息必须被划分成连贯的部分，以便工作小组能够同时工作在不同部分上。即使是一个小系统，人的理解能力也要求将整个模型的内容组织成一个个适当大小的包。包是 UML 模型通用的层次组织单元，它们可以用于存储、访问控制、配置管理以及构造包含可重用的模型单元库。包之间的依赖关系是对包的组成部分之间的依赖关系的归纳。系统整个构架可以在包之间施加依赖关系。因此，包的内容必须符合包的依赖关系和有关的构架要求。
- **扩展机制** 无论一种语言能够提供多么完善的机制，人们总是想扩展它的功能。我们已使 UML 具有一定的扩展能力，相信能够满足大多数对 UML 扩充的需求而不改变语言的基础部分。构造型是一种新的模型元素，与现有的模型元素具有相同的结构，但是加上了一些附加限制，具有新的解释和图标。代码生成器和其他工具对它的处理过程也发生了变化。标记值是一对任意的标记了值的字符串，能够被连接到任何一种模型元素上并代表任何信息，如项目管理信息、代码生成指示信息和构造型所需要的值。标记和值用字符串代表。约束是用某种特定语言（如程序设计语言、专用语言或自然语言）的文本字符串表达的条件。UML 提供了一个表达约束的语言，名为 OCL。与所有其他扩展机制一样，必须小心使用这些扩展机制，因为有可能形成一些别人无法理解的方言；但这些机制可以避免语言基础发生根本性变化。

1.3.3 Delphi 面向对象建模工具 ModelMaker

UML 是一种统一建模语言，通过一组描述手段，它将系统分析、系统设计直至系统程序设计形成有机的联系，UML 的基本设计目标是：

- 提供可视化的模型语言，使得能够设计的模型有统一规范的表达。
- 为扩展核心概念提供了扩展和规范机制。
- 独立于特定的编程语言和开发过程。
- 鼓励面向对象的设计。
- 支持高层次的开发概念，如：合作、框架、模式和组件。

有很多工具可以帮助用户画出 UML 图，但是 UML 建模决不是简单画出几个 UML 图，所以纯粹的绘图工具难以实现真正意义上的 UML 建模功能。真正的对象建模工具是一些能够生成代码的 UML 设计工具，而且需要有正、反向工程的能力，既可以从 UML 图生成代码，又可以从代码生成 UML 图。这样的工具最常用的有 Rational Rose 和 Visio，分别如图 1-7 和图 1-8 所示。不过它们都不支持 Delphi 语言的代码生成。

但是 ModelMaker 是一个与 Delphi 高度集成的、基于 UML 的、面向对象的双向 CASE 工具，用于面向对象系统建模和代码生成，如图 1-9 所示。所谓双向，是指在 ModelMaker 中，只要绘制出 UML 模型图，就可以自动地将其转化为 Delphi 代码；反之，对代码的修改也会在模型中反映出来。这样，用户可以非常方便地在代码和模型之间相互切换。同时，ModelMaker 和 Delphi 的集成程度相当高，对 Delphi 的各种语言特征，如属性、存取方法、单元、VCL 类库等都提供了完全的支持，这也是其他工具所无法比拟的。

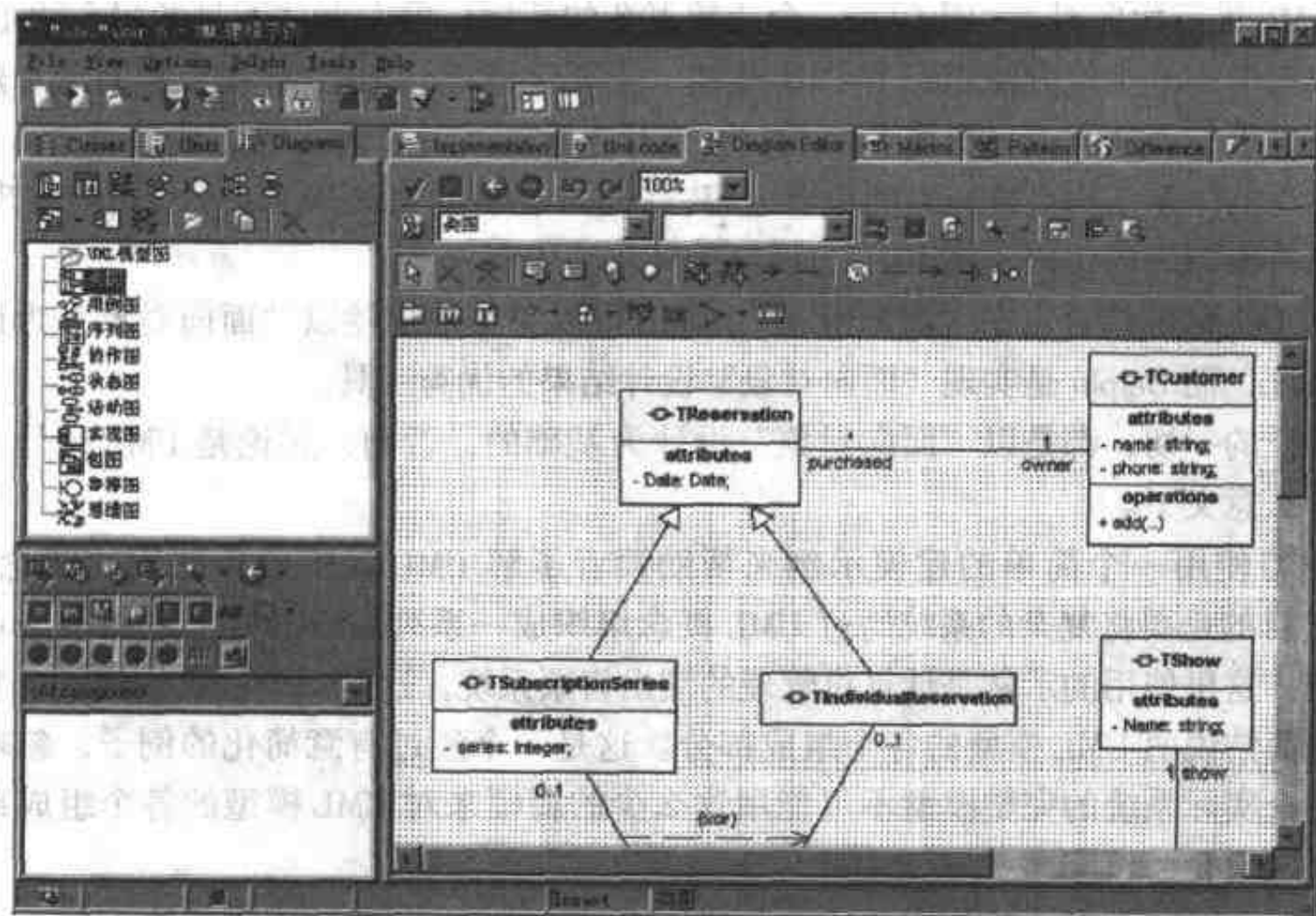


图 1-9 运行中的 ModelMaker

ModelMaker 由一系列工具构成，包括：

- 动态建模引擎，用于存储类之间和其与成员之间的关系，使得对类或其基类的改动会立即传播到自动生成的代码中。
- 模型引入和导出工具，用于在 ModelMaker 模型和源代码中相互转换。
- UML 图生成器，用于将设计可视化。
- 修改单元、类、UML 图、源代码和其他设计特征的专门编辑器。
- 文档化工具，用于简化开发与 MS WinHelp 兼容的在线帮助文件。

必须指出，尽管 ModelMaker 本身不是很大，但功能十分强大。通过对象模型和代码间的高度同步，ModelMaker 为迭代式开发、软件重构、再工程提供了最大的方便；相比之下，大多数 CASE 工具必须通过反复导入和导出模型来完成相应的功能。ModelMaker 同时提供了对设计模式的支持。一系列模式以便于使用的主动代理方式提供，用户可以很方便地根据不同应用，对已有模式进行调整；模式不仅能通过插入 Delphi 代码片断实现某一模式，而且在设计中的改动也能反映到这些代码中，它最大限度地实现了重用。

在后面接下来的 UML 建模示例中，我将用 ModelMaker 绘制的 UML 图来介绍面向对象建模过程。

参见 关于如何使用 ModelMaker，读者可以进一步参阅本书附录 A 的相关内容。

1.3.4 UML 建模示例 (ModelMaker 实现)

使用 UML 进行面向对象的建模是一个比较复杂的话题，但本书并不是专门介绍 UML 建模的读物。我在书中不少地方使用到了 UML 建模，主要是考虑到 UML 有利于帮助读者建立规范的面向对象思维的表达方式。更重要的是，对于有经验的读者来说，UML 建模是 Delphi 7 新提出的模型驱动的体系结构 (MDA) 的重要内容，Delphi 提供的 ModelMaker 不仅可以实现 UML 建模，还可以将模型变成代码，将 UML 建模直接转化为生产力。

UML 与 Delphi 的结合点在“面向对象”，UML 可以完整地描述以“面向对象”为设计思想面设计的系统，而 Delphi 是实现“面向对象”设计结果的优秀工具。

这里所说的一切，都是以“面向对象”设计为基础的，否则，无论是 UML 还是 Delphi 都变得没有什么意义了。

本节我将使用一个简单的建模示例来帮助读者了解 UML 建模中所使用的概念和视图。UML 建模的目的是要将复杂的系统通过 UML 概念组织成一系列较小的视图和图表来可视化描述这个系统。这里使用的示例是计算机管理的演出售票系统。这是一个经典的例子，它可以用少量篇幅来简洁说明 UML 模型的各个组成部分。这是一个经过有意简化的例子，忽略了有关细节，而一个实际系统的完整模型不可能用这么少的篇幅来对 UML 模型的各个组成部分进行介绍。

特别针对 Delphi 开发，我使用的这个示例是用 ModelMaker 6.2 建模实现的。^①

1. UML 视图

UML 中的各种组件和概念之间没有明显的划分界限，但为方便起见，我们用视图来划分这些概念和组件。视图只是表达系统某一方面特征的 UML 建模组件的子集。在每一类视图中使用一种或两种特定的图来可视化地表示视图中的各种概念。

视图分成三个视图域：结构分类、动态行为和模型管理。

结构分类描述了系统中的结构成员及其相互关系。它使用包括类、用例、组件和节点等的类元。类元为研究系统动态行为奠定了基础。类元视图包括静态视图、用例视图和实现视图。

动态行为描述了系统随时间变化的行为。行为用从静态视图中抽取的系统的瞬间值的变化来描述。动态行为视图包括状态视图、活动视图和交互视图。

模型管理说明了模型的分层组织结构。包是模型的基本组织单元。特殊的包还包括模型和子系统。模型管理视图跨越了其他视图，并根据系统开发和配置组织这些视图。

UML 还包括多种具有扩展能力的组件，这些扩展能力有限但很有用。这些组件包括约束、

^① 本书使用的是 Delphi 7 企业版提供的 ModelMaker 6.2，与其他版本 Delphi 兼容的 ModelMaker 可以到 ModelMaker 的官方网站下载。

构造型和标记值，它们适用于所有的视图元素。

表 1-1 列出了 UML 的视图和视图所包括的图以及与每种图有关的主要概念。不能把这张表看成是一套死板的规则，应将其视为对 UML 常规使用方法的指导，因为 UML 允许使用混合视图。实际上不同的 CASE 工具在提供 UML 图时也有自己的差异。

表 1-1 UML 视图和图

主要的域	视图	图	主要概念
结构	静态视图	类图	类、关联、泛化、依赖关系、实现、接口
	用例视图	用例图	用例、参与者、关联、扩展、包括、用例泛化
	实现视图	组件图	组件、接口、依赖关系、实现
	部署视图	部署图	节点、组件、依赖关系、位置
动态	状态视图	状态图	状态、事件、转换、动作
	活动视图	活动图	状态、活动、完成转换、分叉、结合
	交互视图	序列图	交互、对象、消息、激活
		协作图	协作、交互、协作角色、消息
模型管理	模型管理视图	类图	包、子系统、模型
可扩展性	所有	所有	约束、构造型、标记值

ModelMaker 为 Delphi 的面向对象建模提供一组 UML 设计视图，用以描述系统模型，它包括以下内容：

- 用例图 (use case diagram) ——用于表达系统中各个角色及角色间的关系。
- 序列图 (sequence diagram) ——用于表达系统中各种对象的时序安排及其交互过程。
- 类图 (class diagram) ——描述一组对象的类图。
- 状态图 (statechart diagram) ——描述对象状态的变迁，包括：对象可处于什么状态，触发状态变迁的事件。
- 协作图 (collaboration diagram) ——描述为执行某种功能，对象之间需传递什么信息及具体的信息内容。
- 活动图 (activity diagram) ——类似于数据流程图，用于描述操作过程及其判断条件。
- 实现图 (implementation diagram) ——描述系统组件间的交互作用和部署关系。

ModelMaker 将组件图和部署图合二为一。

- 包图 (package diagram) ——描述系统模型自身的组织结构。包是操作模型内容、存取控制和配置管理的基本单元。

下面结合 ModelMaker 的这几种 UML 设计视图讲解建模示例。

2. 用例图

用例图是作为参与者的外部用户所能观察到的系统功能的模型图。用例是系统中的一个功能单元，可以描述为参与者与系统之间的一次交互作用。用例模型的用途是列出系统中的用例和参与者，并显示哪个参与者参与了哪个用例的执行。

图 1-10 是售票系统的用例图。参与者包括售票员、监督员和自动售票机。自动售票机是另一个系统，它接受顾客的买票请求。在售票中心的应用模型中，顾客不是参与者，因为顾客不直接与售票中心打交道。用例包括通过自动售票机或售票员买票、订票（只能通过售票员）

以及售票监督（应监督员的要求）。买票和订票包括一个共同的部分——即通过信用卡来付钱。当然，对售票系统的完整描述还要包括其他一些用例，例如换票和验票等。

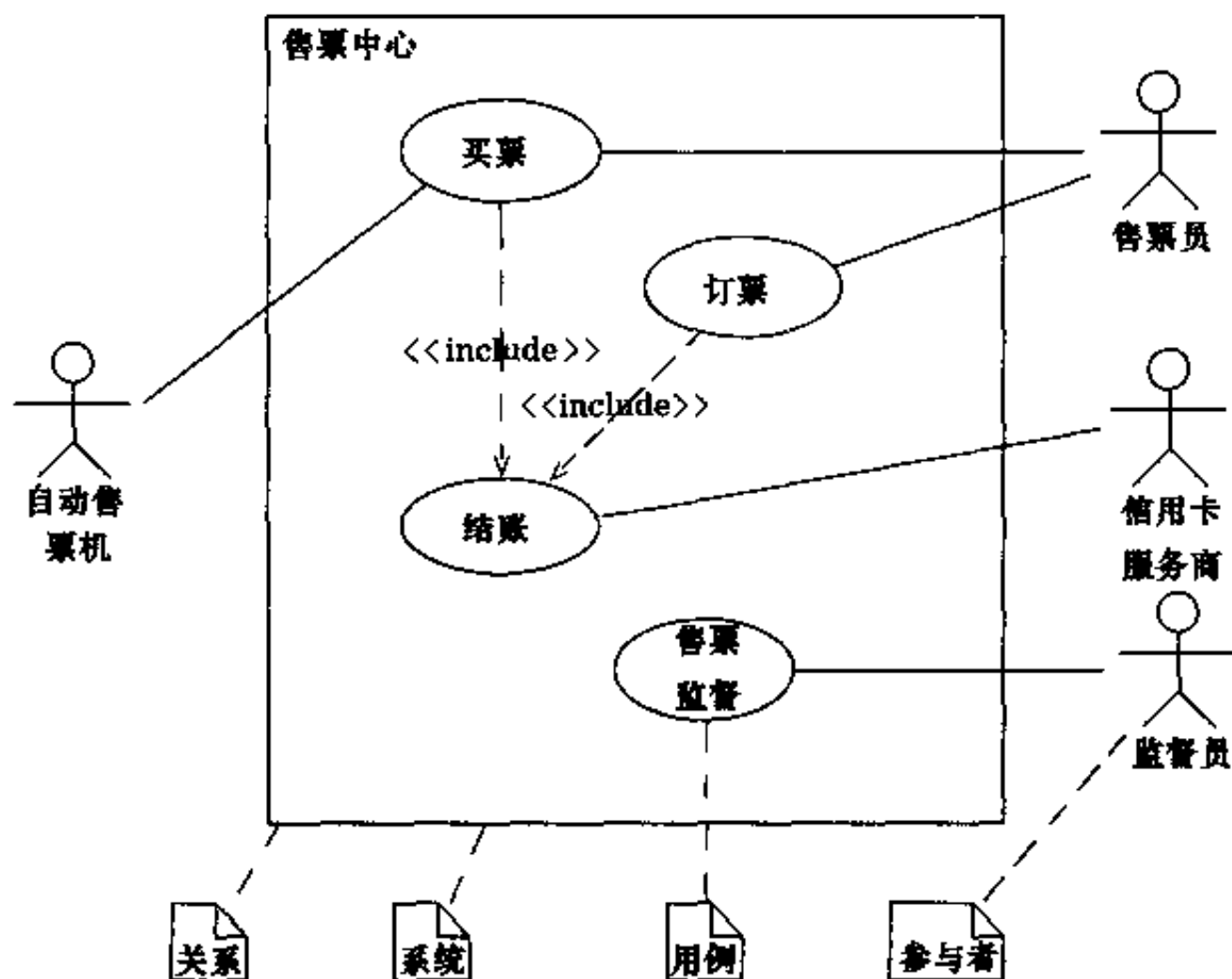


图 1-10 售票系统的用例图

用例也可以有不同的层次。一个用例可以用其他更简单的用例进行说明。在交互视图中，用例作为交互图中的一次协作来实现。

3. 类图

通常使用静态视图对应用领域中的概念以及与系统实现有关的内部概念建模。这种视图之所以被称之为是静态的，是因为它不描述与时间有关的系统行为，此种行为在其他视图中进行描述。

静态视图主要是由类及类间相互关系构成，这些相互关系包括：关联、泛化和各种依赖关系，如使用和实现关系。类是应用领域或应用解决方案中概念的描述。类图是以类为中心来组织的，类图中的其他元素或属于某个类或与类相关联。静态视图用类图来实现，正因为它以类为中心，所以称其为类图。

在类图中，类用矩形框来表示，它的属性和操作分别列在分格中。如不需要表达详细信息时，分格可以省略。一个类可能出现在好几个图中。同一个类的属性和操作可只在一种图中列出，在其他图中可省略。

关系用类框之间的连线来表示，不同的关系用连线上和连线端头处的修饰符来区别。

图 1-11 是售票系统的类图，它只是售票系统领域模型的一部分。图中表示了几个重要的类，如 TCustomer、TReservation、TTicket 和 TPerformance。一个顾客可多次订票，但每一次订票只能由一个顾客来执行。有两种订票方式：个人票或套票，前者只是一张票，后者包括多张票。

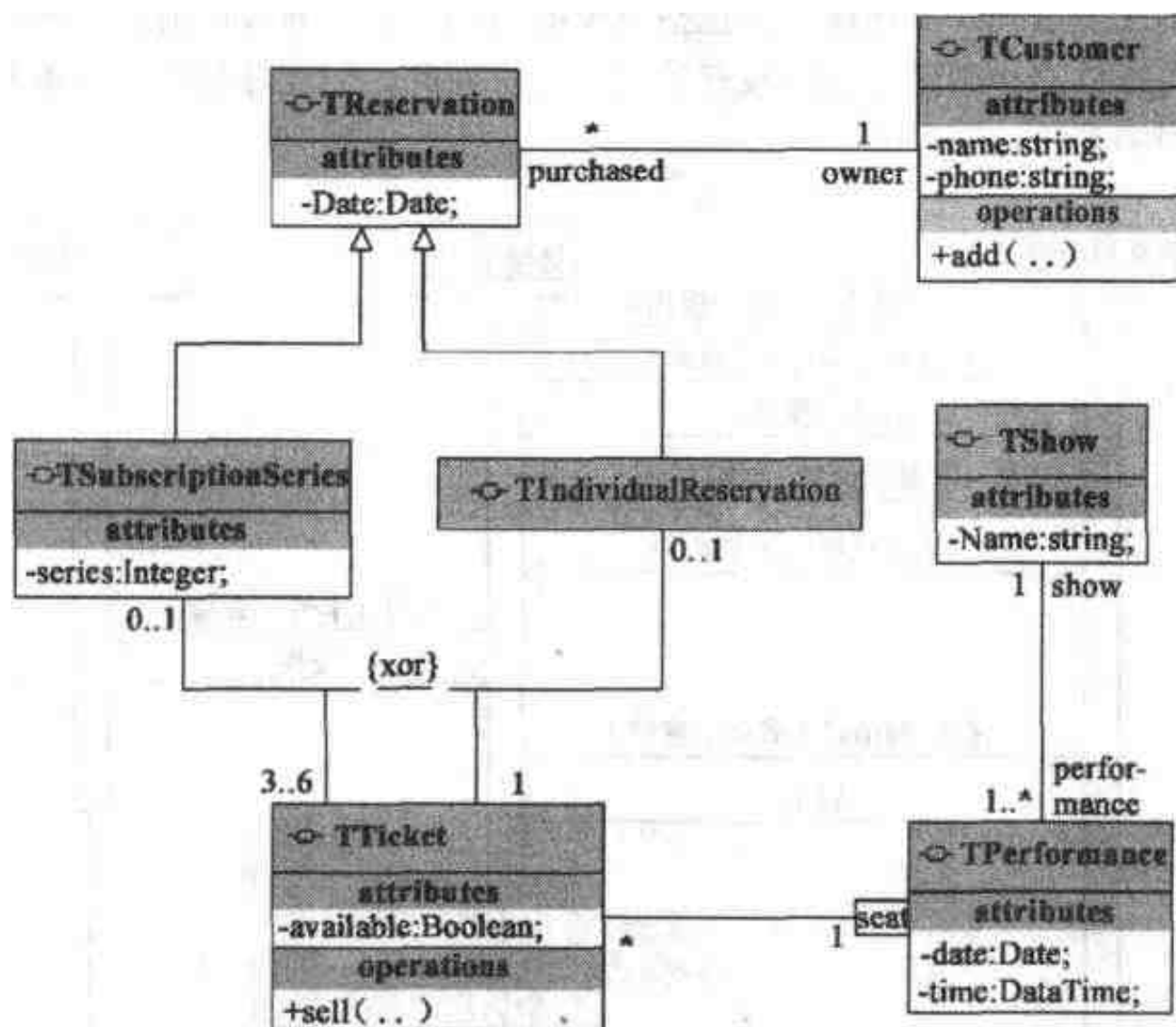


图 1-11 售票系统的类图

每一张票不是个人票就是套票中的一张，但是不能既是个人票又是套票中的一张。每场演出都有多张票可供预定，每张票对应一个惟一的座位号。每次演出用剧目名、日期和时间来标识。

类可用不同的精确度和抽象级别来描述。在设计的前期阶段，所建立的模型只是问题的逻辑模型，到了设计的后期，模型中会增添许多设计结论和有关系统实现的细节。UML 中的大部分视图都可不断进行类似的细化。

4. 序列图和协作图

交互视图描述了执行系统功能的各个角色之间相互传递消息的顺序关系。类元是对在系统内交互关系中起特定作用的一个对象的描述，这使它区别于同类的其他对象。交互视图显示了跨越多个对象的系统控制流程。交互视图可用两种图来表示：序列图和协作图，它们各有不同的侧重点。

序列图表示了对象之间传送消息的时间顺序。每一个类元角色用一条生命线来表示，即用垂直线代表整个交互过程中对象的生命期。生命线之间的箭头连线代表消息。序列图可以用来进行一个场景说明，即一个事务的历史过程。

序列图的一个用途是用来表示用例中的行为顺序。当执行一个用例行为时，序列图中的每条消息对应了一个类操作或状态中引起转换的触发事件。

图 1-12 是描述购票这个用例的序列图。顾客在自动售票机与售票中心通话触发了这个用例的执行。序列图中付款这个用例包括售票中心与自动售票机和信用卡服务处的两个通信过

程。这个序列图用于系统开发初期，未包括完整的与用户之间的接口信息。例如，座位是怎样排列的；对各类座位的详细说明都还没有确定。尽管如此，交互过程中最基本的通信已经在这个用例的序列图中表达出来了。

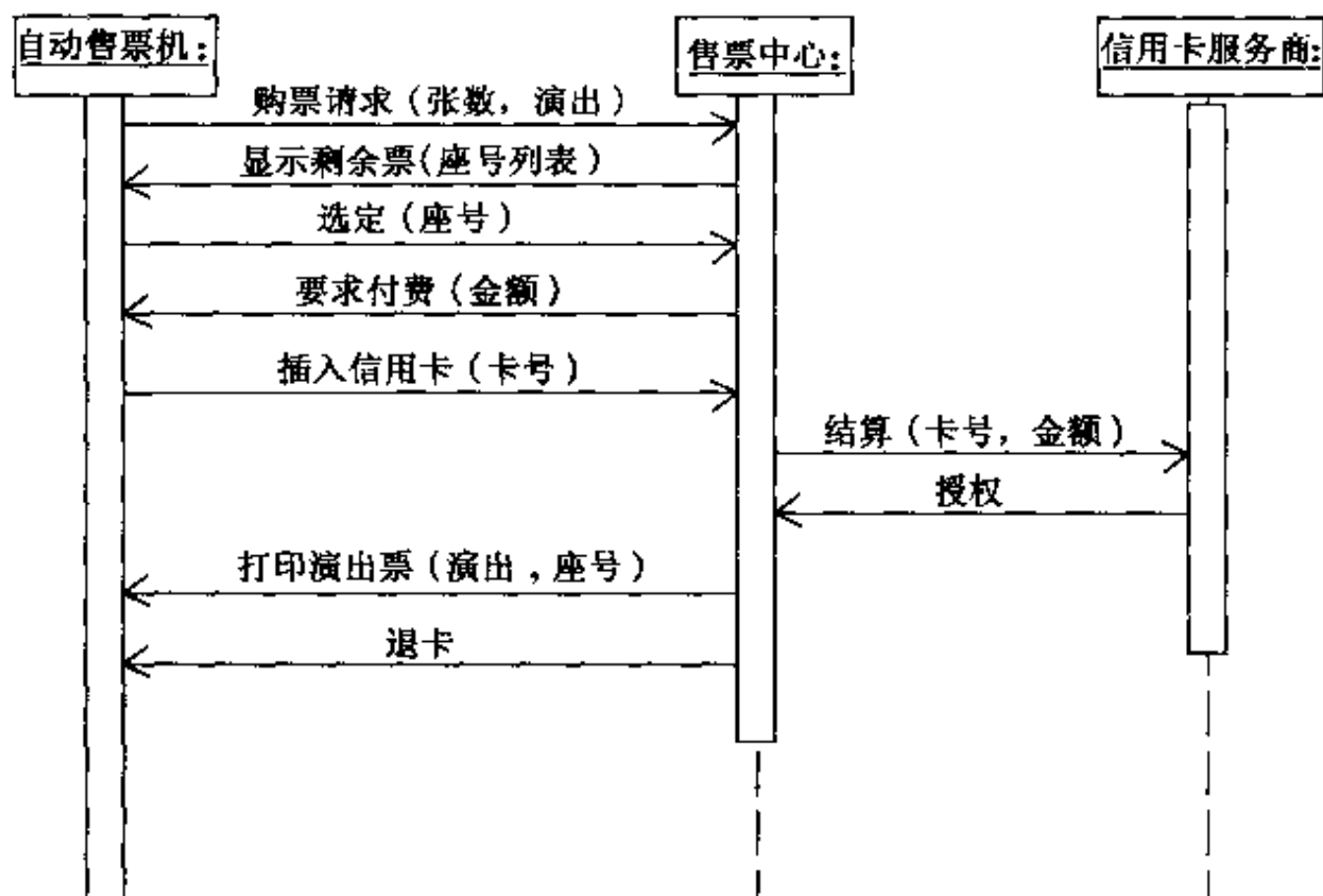


图 1-12 描述购票用例的序列图

协作图对在一次交互中有意义的对象和对象间的链建模。对象和关系只有在交互的语境中才有意义。类元角色描述了一个对象，关联角色描述了协作关系中的一个链。协作图用几何排列来表示交互作用中的各角色，如图 1-13 所示。附在类元角色上的箭头代表消息。消息的发生顺序用消息箭头处的编号来说明。

协作图的一个用途是表示一个类操作的实现。协作图可以说明类操作中用到的参数和局部变量以及操作中的永久链。当实现一个行为时，消息编号对应了程序中嵌套调用结构和信号传递过程。

图 1-13 是开发过程后期订票交互的协作图。这个图表示了订票涉及的各个对象间的交互关系。请求从自动售票机发出，要求从所有的演出中查找本次演出的资料。返回给引用 ticket seller 对象的指针 db 代表了与某次演出资料的局部暂时链接，这个链接在交互过程中保持，交互结束时释放。售票方准备了许多演出的票；顾客在各种价位做一次选择，锁定所选座位，售票员将顾客的选择返回给自动售票机。当顾客在座位表中做出选择后，所选座位号被标记为已购买的座位号，其余座位解锁。

序列图和协作图都可以表示各对象间的交互关系，但它们的侧重点不同。序列图用消息的几何排列关系来表达消息的时间顺序，各角色之间的相关关系是隐含的。协作图用各个角色的几何排列图形来表示角色之间的关系，并用消息来说明这些关系。在实际中可以根据需要选用这两种图。

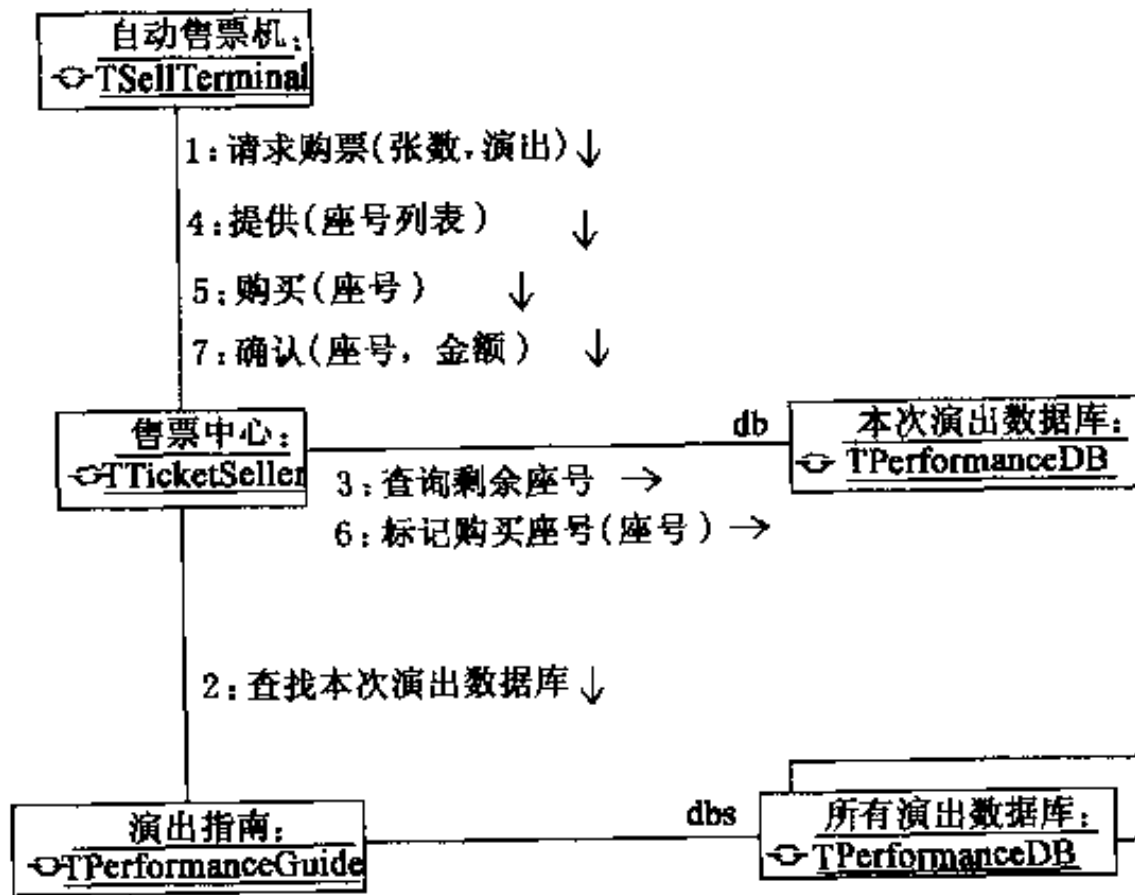


图 1-13 订票交互过程的协作图

5. 状态图

状态图是一个类对象所可能经历的所有历程的模型图。状态图由对象的各个状态和连接这些状态的转换组成。每个状态对一个对象在其生命期中满足某种条件的一个时间段建模。当一个事件发生时，它会触发状态间的转换，导致对象从一种状态转化到另一新的状态。与转换相关的活动执行时，转换也同时发生。状态用状态图来表达。

图 1-14 是票这一对象的状态图。初始状态是 Available 状态。在票开始对外出售前，一部分票是给预约者预留的。当顾客预定个人票时，被预定的票首先处于锁定状态，此时顾客仍有确定是否购买这张票的选择权，故这张票可能出售给顾客也可能因为顾客不要这张票而解除锁定状态。如果超过了指定的期限顾客仍未做出选择，此票将被自动解除锁定状态。预约者也可以换其他演出的票，如果这样的话，最初的预约票也可以对外出售。

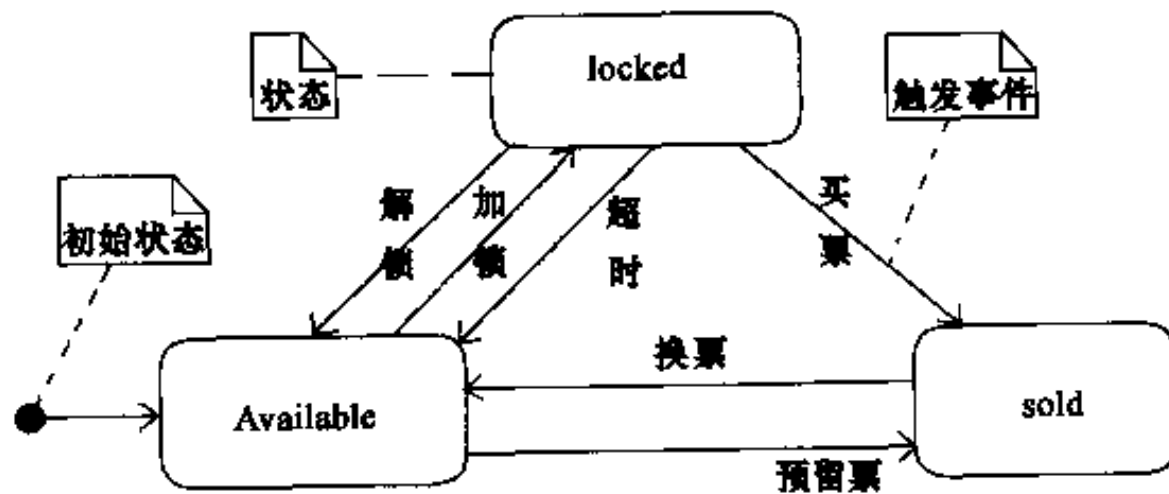


图 1-14 票对象的状态图

状态图可用于描述用户接口、设备控制器和其他具有反馈的于系统。它还可用于描述在生命期中跨越多个不同性质阶段的被动对象的行为，在每一阶段，该对象都有自己特殊的行为。

6. 活动图

活动图是状态的一个变体，用来描述执行算法的工作流程中涉及的活动。活动状态代表了一个活动：一个 workflow 步骤或一个操作的执行。活动图描述了一组顺序的或并发的活动。

图 1-15 是售票中心的活动图。它表示了上演一个剧目所要进行的活动（这个例子仅供参考，不必太认真地凭着看戏的经验而把问题复杂化）。箭头说明活动间的顺序依赖关系——例如，在制定演出计划前，首先要选择演出的剧目。加粗的横线段表示分叉和结合控制。例如，安排好整个剧目的进度后，可以进行宣传报道、购买剧本、雇用演员、准备布景、设计照明、设计戏服等，所有这些活动都可同时进行。在进行彩排之前，必须已经具备了剧本和演员。

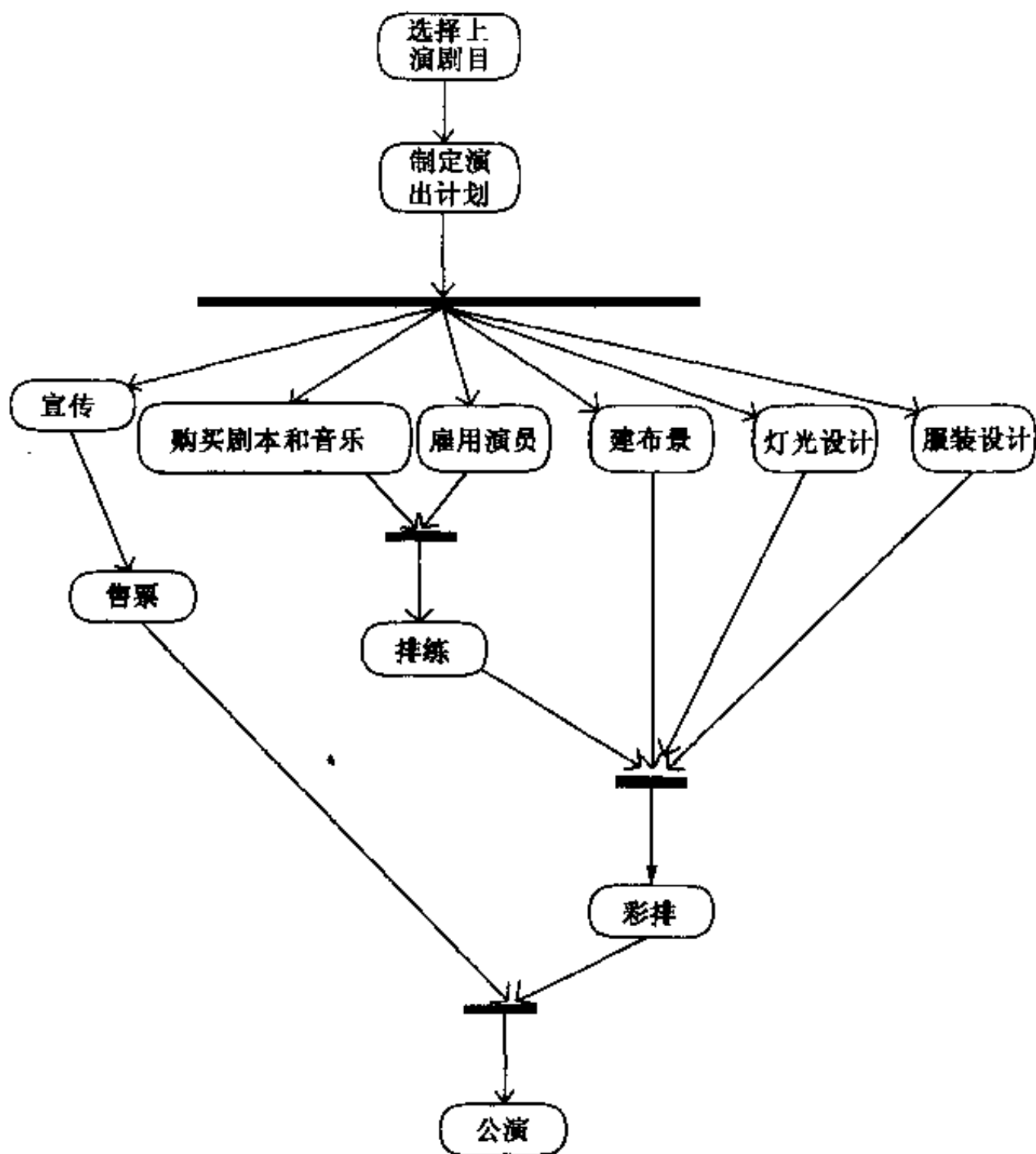


图 1-15 售票中心的活动图

这个例子说明活动图的用途是对人类组织的现实世界中的工作流程建模。对事物建模是活动图的主要用途，但活动图也可对软件系统中的活动建模。活动图有助于理解系统高层活动的执行行为，而不涉及建立协作图所必须的消息传送细节。

在图 1-15 中我们可以看到，活动图用连接活动和对象流状态的关系流表示活动所需的输

入输出参数。

7. 实现图

前面介绍的视图模型按照逻辑观点对应用领域中的概念建模。物理视图对应用自身的实现结构建模，例如系统的组件组织和建立在运行节点上的配置。这类视图提供了将系统中的类映射成物理组件和节点的机制。物理视图有两种：组件图和部署图。在 ModelMaker 中，将组件图和部署图合二为一成为实现图。

实现图用于系统的组件建模。组件是构造应用的软件单元，实现图不仅包含组件还包括各组件之间的依赖关系，以便通过这些依赖关系来估计对系统组件的修改给系统可能带来的影响。图 1-16 是售票系统的实现图。售票方组件顺序接受来自售票员和自动售票机的请求；信用卡主管组件用于处理信用卡付款；还有一个存储票信息的数据库组件。组件图表示了系统中的各种组件。在个别系统的实际物理配置中，可能有某个组件的多个备份。

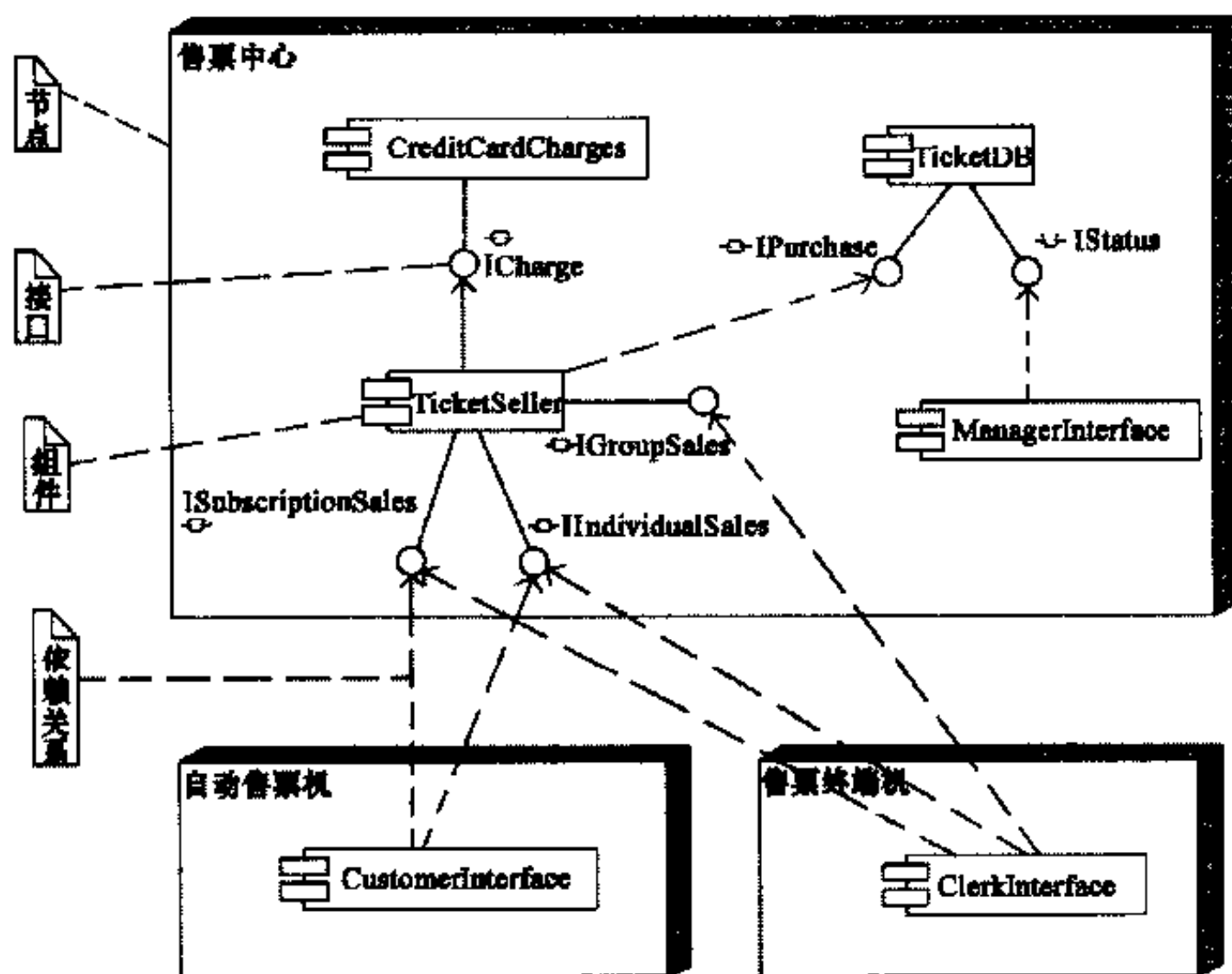


图 1-16 售票系统的实现图

图中的小圆圈代表接口，即服务的连贯集。从组件到接口的实线表明该组件提供的列在接口旁的服务。从组件到接口的虚线箭头说明这个组件要求接口提供的服务。例如，购买个人票可以通过自动售票机订购也可直接向售票员购买，但购买团体票则只能通过售票员进行。

实现图还可以描述位于节点实例上的运行组件实例的安排。节点是一组运行资源，如计算机、设备或存储器。这个视图允许评估分配结果和资源分配。

这就是说，部署视图可以用实现图来表达。图 1-16 也描述了售票系统的部署。图中表示了系统中的各组件和每个节点包含的组件，以及所有这些组件和它们之间的连接。节点用立方

体图形表示。

8. 包图

包图是一种模型管理视图，用于对模型自身组织建模。一系列由模型元素（如类、用例）构成的包组成了模型。包可能包含其他的包，因此，整个模型实际上可看成一个根包，它间接包含了模型中的所有内容。包是操作模型内容、存取控制和配置控制的基本单元。每一个模型元素包含于包中或包含于其他模型元素中。

模型是从某一观点以一定的精确程度对系统所进行的完整描述。从不同的视角出发，对同一系统可能会建立多个模型，例如有系统分析模型和系统设计模型之分。模型是一种特殊的包。

子系统是另一种特殊的包。它代表了系统的一个部分，它有清晰的接口，这个接口可作为一个单独的构件来实现。包的详细信息通常在类图中表达。

图 1-17 显示了将整个系统分解所得到的包和它们之间的依赖关系。售票中心子系统在前面的例子中已经讨论过了，完整的系统还包括营运管理和计划子系统。每个子系统还包含了多个包。

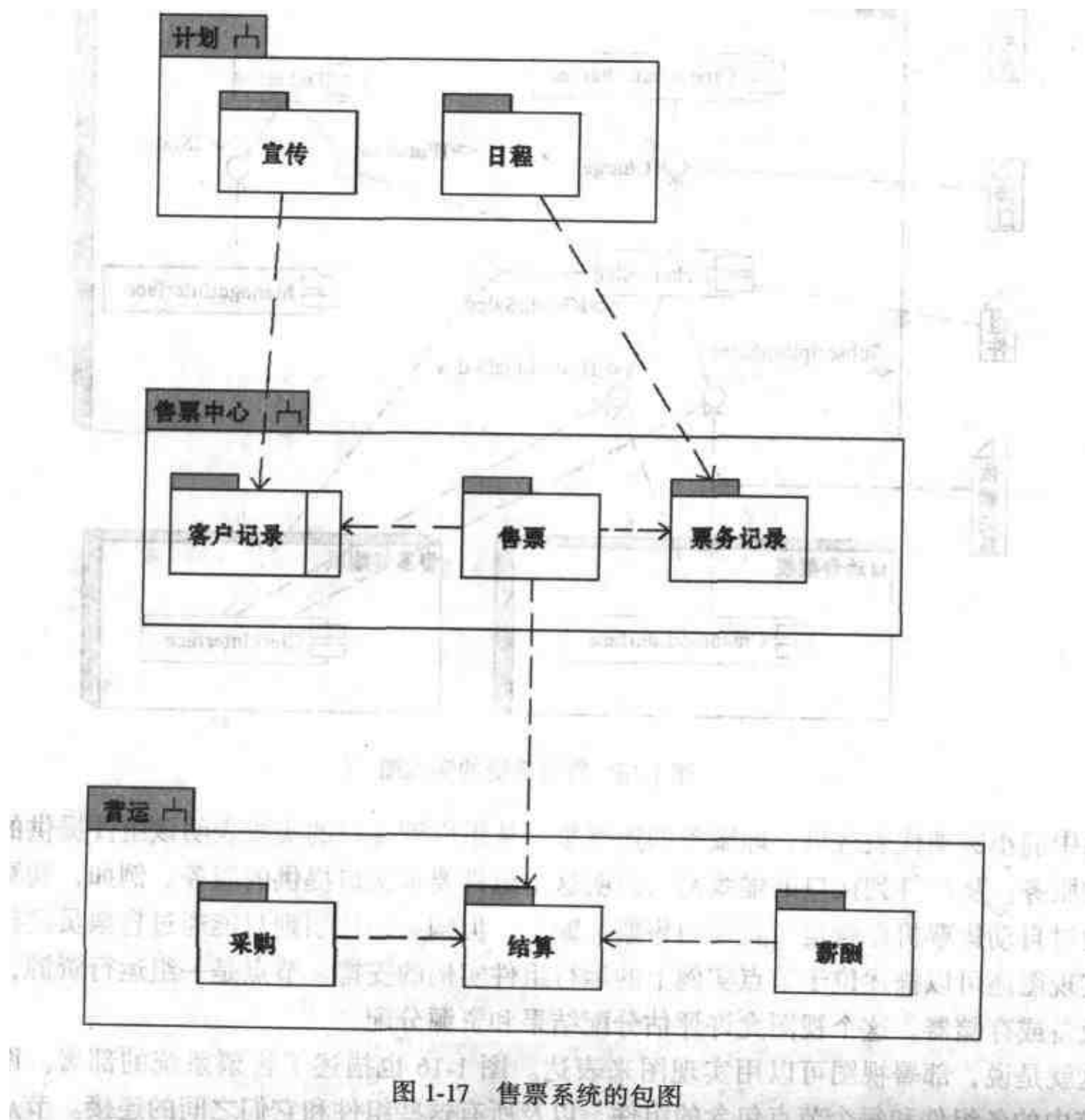


图 1-17 售票系统的包图

9. 各种视图间的关系

通常多个视图共存于一个模型中，它们的元素之间有很多关系，其中一些关系列在表 1-2 中。表中没有将各种关系列全，但它列出了从不同视角观察得到的元素间的部分主要关系。所以，UML 建模实际上是通过多个 UML 视图以及其反映出的不同元素的关系来定义和描述模型的。这种建模工具尤其适用于离散的、面向对象的模型。

表 1-2 不同视图元素间的部分关系

元 素	元 素	关 系
类	状态	拥有
操作	交互	实现
用例	协作	实现
用例	交互实例	样本场景
组件实例	节点实例	位置
动作	操作	调用
动作	信号	发送
活动	操作	调用
消息	动作	激发
包	类	拥有
角色	类	分类

参见 要进一步了解 ModelMaker 的详细用法，可以参见本书附录。

第 2 章 Delphi 对象模型

2.1 类和对象

类和对象并不是一回事。类是一种类型的定义，而对象则是这种叫做“类”的类型的实例。在 Delphi 中有很多种数据类型，如：Integer、Record 等类型。而类则是一种用户定义的数据类型，它具有自己的说明和一些操作。类中含有一些内部数据和一些过程或函数形式的对象方法，通常来描述一些非常相似的对象所具有的共同特征和行为。

类的定义在一些面向对象的书上可能比较晦涩。你可以把类想像为一种特殊的 Record 类型，其中不但可能包含数据，而且还可能包含函数和过程（在 OOP 中称之为方法）。这些数据和方法被统称为类的成员。

一旦创建了一个类类型，就可以生成基于该类的对象，且对象的数量可以无限。这就好像用一个做饼干的模具来生产大量的同一种饼干一样。做饼干的模具相当于类，而生产出的饼干则是对象——饼干的实例，也就是具体的某一块饼干。

2.1.1 类

我们已经知道类的概念就是，抓住对象的相似性，定义它们的共同特征，包括数据和操作。类声明是一种类型声明，在 Delphi 中类被当做一个类型来定义，它的语法是：

```
type 类名 = class (基类)
    {数据成员声明}
    {过程和函数声明}
    {属性的声明}
end;
```

类声明描述了类的数据成员、方法和属性。可以在一个单元的接口或者实现部分声明一个类，但方法（如同任何其他函数或过程）是定义在实现部分的。必须在与类声明相同的单元内实现类的方法。

类声明具有一个或多个不同存取级别的部分，它们是 private（私有的）、protected（受保护的）、public（公有的）、published（公布的）和 automated（自动的）。存取级别也称为类成员的可见性，将在 2.3 节讨论。

类可以有任何数目的数据成员，后面跟有方法和属性的声明。方法和属性的声明可以混合在一起，但在每一个部分之中，所有的数据成员必须在所有的方法和属性之前。与 Delphi 和 C++ 不同，不能在一个类声明中声明任何嵌套的类型。一个类具有一个基类（也称父类），它从中继承所有的数据成员、属性和方法。如果不列出一个明确的基类，Delphi 将使用 TObject。类可以实现任意数目的接口。

注意 Delphi 中的惯例是类型名称以 T 开头，如 TObject。这仅仅是一个惯例，而不是

语言规则。另一方面，IDE 总是以 T 开头对类进行命名。

示例程序 2-1 展示了一个类声明的例子。

示例程序 2-1 一个 TMan 的类声明

```
type
  TMan = class (TObject)
  private
    FAge: Integer;
    procedure SetAge (Value: Integer);
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; overload;
    property Age: Integer read FAge write SetAge;
    procedure sayHello (words: pchar);
  end;
```

2.1.2 类成员

在类声明中，我们注意到一个类中包含了数据成员（Field）、方法（Method）和属性（Property）。通常我们把类的数据成员、方法和属性称为类成员。

数据成员（也称字段、域），它们是一些在类中声明的定义好的变量。虽然不少 Delphi 的书中将 Field 译为“字段”，但由于字段的叫法容易造成理解上的混乱（比如数据库中也有字段），因此本书统一使用数据成员的提法。这样也能和 Java、C++ 等其他面向对象语言中的叫法一致。

方法则是一些封装在类中的过程和函数，用于执行类的操作，完成类的任务。

属性通常用做访问对象数据的接口，对象的数据一般存储在数据成员中。属性有存取设定，它决定数据如何被读取和修改。从程序的其他地方（在对象本身之外）来看，属性在很大程度上很像一个数据成员，但本质上它更像方法，比如在类的实例（对象）中并不为它分配内存。可以这么说，这种叫做属性的方法是用于对私有数据成员进行读写操作的，但在使用时与直接使用数据成员一样。不妨可以把它看做是一种含有约束机制的数据成员。后面我会详细介绍属性和数据成员相比，在使用上的优点。

在示例程序 2-1 中，TMan 类的数据成员有：Name、Language、SkinColor 等，用于存储人的姓名、语言和肤色等信息；属性有 Age，用于年龄的读写；方法有 sayHello 和 SetAge，其中 SetAge 用于对 Age 属性的设置（写），以保证年龄合法（比如：没有负的年龄，或年龄都小于 140）。

通常，使用 UML 类图可以清晰地看到类的成员以及类之间的关系。图 2-1 就是用 Delphi 7 的工具 ModelMaker 制作的一个 UML 类图，其中包含了我们声明的 TMan 类。

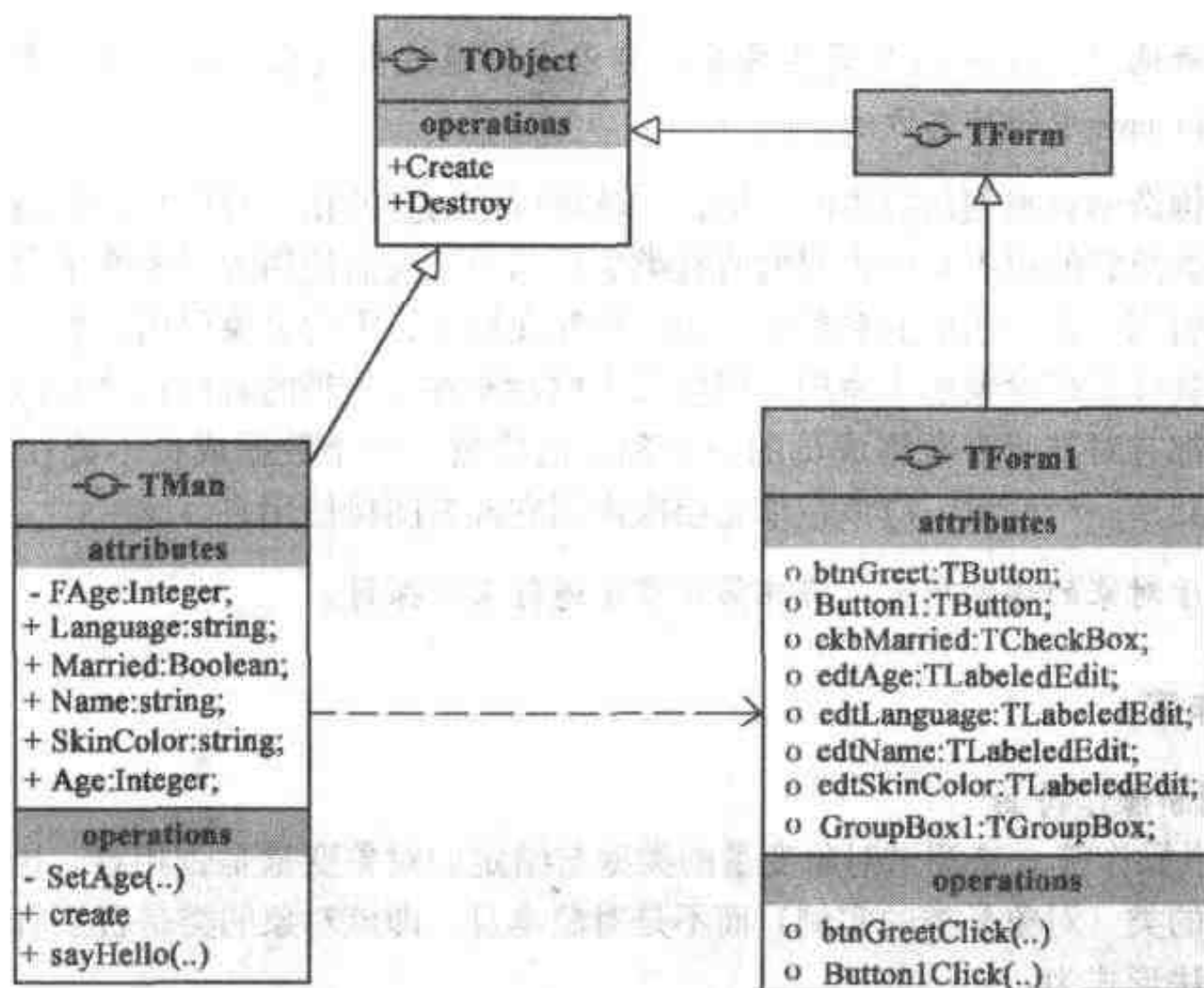


图 2-1 UML 类图

2.1.3 对象

在 Delphi 程序中，光有类还无法使用。必须对类进行实例化，并对类中的方法进行实现。在单元的实现部分，不必按照定义的顺序，但是必须按照以下格式来做：

```
procedure (或 function) 类名. 方法 (参数表);
```

注意 在方法内引用程序其他部分建立的对象，如果运行过程中该对象未建立，访问它时会发生保护性的错误。

Delphi 通过调用类的一个构造函数来建立一个对象的实例，构造函数主要用来为对象创建实例并为对象中的成员分配内存并进行初始化，以使得对象处在可以使用的状态。Delphi 的对象至少有一个称为 Create () 的构造函数，但一个对象可以有多个构造函数。根据不同的对象类型，Create () 可以有不同的参数。与在 C++ 中不同，在 Delphi 中构造函数不能自动调用，程序员必须自己调用构造函数，调用构造函数的语法如下：

```
MyObject := TMyObject.Create;
```

注意这里调用构造函数的语法有点特殊，是通过类型来引用一个对象的 Create () 方法，而不是像其他方法那样通过实例来引用。这看上去有点奇怪，但很有意义。变量 MyObject 在调用时还没有定义，而类 TMyObject 已经静态地存在于内存中，静态调用它的 Create () 方法是合法的。

注意 准确地讲, Create () 是类方法, 可以直接通过类 (无需实例化) 来调用。相当于 C# 和 Java 中的静态方法。

通过调用构造函数来创建对象的实例, 这就是所谓的实例化。对象是类的动态实例。动态实例包含了在类及它的祖先类中声明的所有类成员的值。我们使用的对象变量实际上是对对象的一个引用。该引用保存在栈中而对象总是动态分配在堆上, 因此对象引用是真正指向对象的指针。程序员有责任管理对象的生命期, 即负责生成对象并在适当的时候释放它们。

每个对象都有对其所有数据成员的一个独立的拷贝。一个数据成员不能在多个对象中共享。如果需要共享一个变量, 应该在单元层次声明它或者间接使用它。

参见 关于对象的详细内容, 将在第 3 章中进行深入探讨。

2.1.4 类操作符

1. 类型判断操作符 is

Delphi 提供操作符 is 来测试对象变量的类型与给定的对象变量是否相容。也就是说, is 是用于判定对象的类 (对象是类的实例) 而不是对象本身, 即该对象的类是否是右操作数的同类或派生类。语法形式为:

对象变量 is 类名;

在 Windows 程序设计中常用此运算判断控件的类型, 如:

```
procedure TForm1.ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;  
    State: TDragState; var Accept: Boolean);  
begin  
    Accept := (Source is TEdit);  
end;
```

该程序段用于判断拖放到列表框 ListBox1 中的是不是文本框 (TEdit), 如果是该类型的控件, 就接受拖放操作。具体程序参见后面的示例程序 4-13。

2. 类型强制转换符 as

as 操作符把一个对象变量的类型转换为另一个类的类型或者把一个接口变量转换为另一个接口的类型, 表达式的类型是 as 运算符右边的类和接口的类型。语法形式为:

对象变量 as 类名;

如果对象或接口变量是 nil, 则结果是 nil。欲转换对象所声明的类必须是目标类的后代或祖先 (即有血缘关系), 否则类类型不兼容将导致转换失败。as 进行类型强制转换时首先测试, 然后进行转换, 转换不成功, 则引起异常 EInvalidCast。为了安全起见, 有时需要使用 is 操作符判定对象的类型, 避免转换出错。

```
procedure TForm1.ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);  
begin  
    if Source is TEdit then  
        ListBox1.Items.Add ((Source as TEdit).Text);  
    if Source is TLabel then
```

```
ListBox1.Items.Add ( (Source as TLabel) .Caption);  
end;
```

上面的示例程序段用于判断拖放到列表框 ListBox1 中的控件类型，如果是文本框 (TEdit)，就将拖放控件强制转换为 TEdit 类型，在列表框中添加它的 Text 属性值；如果是标签 (TLabel)，就将拖放控件强制转换为 Label 类型，在列表框中添加它的 Caption 属性值。具体程序参见后面示例程序 13。

3. 隐含参数 Self

在每一个方法中，Delphi 声明 Self 变量为一个隐含参数。在普通方法中，Self 变量的值是对象的引用；在一个类方法中，Self 变量的值是类的引用。

Self 是一个隐含的参数，是表示自身的变量，调用对象方法时，该参数便被传递了过来。这相当于每个方法体中都有一个隐式的“with self do”。也就是说，所有的数据成员、方法和属性在作用域中，无需显式地引用 Self 就可以访问它们。

它常被用于以下三种情况：

- 如果在方法内调用另外一个对象的方法时，需要将自身参数传递过去，那么把 Self 传递过去好了。
- 如果派生类和基类出现了数据成员的重名，想要访问基类的数据成员，而每次只能得到派生类本身的重名数据成员，怎么办？使用强制转换“基类名 (Self) . 数据成员”。
- 如果在方法内调用了与数据成员同名的局部变量，则使用变量名字只能访问到局部变量，使用“Self . 变量名”便访问到了数据成员。

2.2 方法

2.2.1 什么是方法

方法是在类中定义，用来实现对象操作的过程或者函数。方法是属于一个给定对象的过程和函数，方法反映的是对象的行为而不是数据。

方法与普通函数和过程不同，是因为方法只应用于特定类及其祖先类的对象。另外，每一个方法都有一个隐含的参数，称为 Self，它引用作为方法调用主体的对象，这也是普通函数和过程所没有的。

注意 在类方法中，Self 不引用对象，而是引用类。

调用方法和调用普通函数或过程相同，但要以对象引用作为方法名称的开头，例如：

```
MyObject.Method (Argument );
```

方法可以应用于类或通过继承作用于其子类。也就是说，可以调用一个在对象的类或它的任何祖先类中声明的方法。如果相同的方法在祖先类和派生类里面都有声明，则 Delphi 调用派生层次中最顶层的那个方法，如示例程序 2-2 所示。

 示例程序 2-2 调用方法

```

type
  TMan = class (TObject)
  public
    procedure SayHello (w: string); //祖先类声明的方法
  end;

  TChinese = class (TMan)
  public
    constructor create;
    procedure SayHello (w: string); //派生类声明的方法
  end;

var
  AMan: TMan;
  AChinese: TChinese;
begin
  AMan.SayHello ('你好!')      //调用的是 TMan.SayHello 方法
  AChinese.SayHello ('你好!'); //调用的是 TChinese.SayHello 方法
  .....
end;

```

2.2.2 方法的分类

Delphi 中方法的划分没有统一的标准，通常按照用途分，可分为普通方法、构造方法（constructor，也译为构造函数、构造子）、析构方法、类方法和消息处理方法；按照运行绑定机制分，可分成静态方法、虚方法、动态方法。为了让读者有一个整体概念，下面简要介绍一下这些方法。有些方法的详尽使用技巧会在后面的话题中进一步讨论。

1. 普通方法

此类方法是在类内定义、单元内实现的，可以是过程，也可以是函数。方法是类的成员，通过设置保护方式，也可确定它的调用范围。定义普通方法的语法如下：

```

type 类名 = class (基类)
|可见性限定符|
  private
  ...
  |或者 public |

  procedure 方法名 (参数表);
  function 方法名 (参数表): 返回值类型;
end;

```

实现一个方法的语法如下：

```

procedure 类名.方法名 (参数表);
|function 类名.方法名 (参数表): 返回类型;|
  |常量, 变量等定义|
begin
  |执行语句;|
end;

```

调用一个方法的语法格式如下：

对象实例.方法名 (实际参数);

普通方法的实现与子程序的实现在形式上的差别是方法名前面要加上类名的限定。在一般方法的实现中，最好只访问自己类中的成员，不要访问类外的对象，防止对象未建立而发生保护性错误！你可以调用祖先类的方法，但如果在派生类和基类中都定义了一个同名的方法，则派生类的方法则会覆盖（或隐藏）基类的方法。

2. 构造方法

构造方法习惯上称为构造函数，用来创建类的实例（对象），并且对对象的数据成员进行初始化。构造方法之所以被称为构造函数，是因为它返回的是一个类的实例指针，即对象的引用。构造方法很像一般方法，只是将 procedure 换成限定符 constructor。构造方法的格式如下：

```
type 类名 = class (基类)
...
    constructor 构造方法名 (参数表);
end;
```

实现构造方法的语法如下：

```
constructor 类名.构造方法名 (参数表);
```

调用构造方法创建对象实例的语法如下：

```
对象变量名 := 类名.构造方法名 (参数表);
```

示例程序 2-3 声明和实现了 create 构造方法。

示例程序 2-3 使用构造函数

```
type
    TMan = class (TObject)
    public
        Language: string;
        Married: Boolean;
        Name: string;
        SkinColor: string;
    end;

    TChinese = class (TMan)
    public
        constructor create;
        function SayHello: string;
    end;

implementation

constructor TChinese.create;
begin
    Name := '张三';
    Language := '中文';
    SkinColor := '黄色';
end;
```

构造方法创建类的实例，创建后返回该实例的引用。构造方法除了继承基类的构造方法外，一般还创建其他数据结构，还要对类的所有数据成员进行初始化。构造方法不是必须的，如果你不声明自己的构造方法，就默认使用最近祖先类的构造方法。这就是说，如果不想对基类的构造方法进行扩充，就没有必要在程序中声明构造方法。

如果由于某种原因，构造方法执行失败了，对象引用的返回值便是 nil，因此在程序中你有必要对那些对资源要求比较苛刻的构造方法实行保护。如果创建未成功，程序却仍按正常执行，来访问新建对象的数据成员或方法，就会出现保护性错误。

3. 析构方法

析构方法习惯上称为析构函数，用来销毁类的实例（对象），并且销毁对象中的其他数据结构。通常情况下，一个类只有一个析构方法 `destroy`。这个析构方法通常不带任何参数（带参数的析构方法很少），因为目标明确，就是关闭、销毁。析构方法的语法格式如下：

```
type 类名 = class (基类)
...
destructor 析构方法名 (参数表);
end;
```

实现析构方法时按照如下语法：

```
destroy 类名 . 析构方法名 (参数表);
```

调用析构方法的语法如下（一般不直接调用析构方法）：

```
对象变量名 . 析构方法名 (参数表);
```

如果你在构造方法中分配了内存，使用了资源，打开了文件、数据库，此时需要析构方法去做善后工作。析构方法 `destroy` 在实现时要注意三点：

首先，考虑到一个类的析构方法可以不止一个，类声明时建议覆盖继承下来的析构方法，并不再声明其他的析构方法：

```
destructor destroy; override;
```

其次，实现析构方法时按照如下格式，先执行本身的命令，最后继承祖先类的析构方法：

```
destructor TMyclass.destroy;
begin
... {先执行本身的命令,完成销毁、关闭等工作}
inherited destroy; {最后继承祖先类的析构方法}
end;
```

最后，不要在程序中调用 `destroy` 来销毁对象，而应该用 `free`。`free` 在销毁之前会检测对象是否为 nil，如果不为 nil 才销毁对象变量。

4. 类方法

普通方法只能通过对象实例来调用，而类方法既可以通过对象实例调用，也可以通过类本身引用。类方法只是表明这个方法在逻辑上与这个类有联系。类方法在类声明中定义，与普通方法的区别就是在 `procedure` 或者 `function` 之前加一个 `class`。

类方法是作用于类而不是对象上的方法，要考虑到在类方法中使用类名调用时，各数据域都未创建，因此在类方法内，不允许访问类的数据域。实现类方法时按照以下语法：

class procedure (或 function) 类名, 类方法名 (参数表);

调用类方法的语法如下:

类名, 类方法名 (参数表);

或者:

对象实例变量名, 类方法名 (参数表);

最典型的类方法就是 TObject 的方法 ClassInfo、ClassName、ClassParent、InheritsFrom、InstanceSize、MethodAddress、MethodName。这些方法提供了关于该类及其实例的有用信息。如 ClassName 是按照如下方式定义的:

```
class function ClassName: ShortString;
```

5. 消息处理方法

消息处理方法 (即 message 方法) 是用来响应动态分派的消息的。用 message 方法来响应 Windows 的消息, 这样就不用直接来调用它了。

在声明时, 通过包含 message 限定符来创建一个 message 方法, 并在 message 后面跟一个介于 1~49151 之间的整数常量, 它指定消息的编号 (ID)。对于 VCL (Visual Component Library) 控件, message 方法中的整数常量可以是 message 单元中定义的 Windows 消息编号, 这里还定义了相应的记录类型。一个 message 方法必须是具有单一引用参数 (var) 的过程。比如, 在 Windows 下:

```
type
  TTextBox = class (TCustomControl)
  private
    procedure WMChar (var Message: TWMChar); message WM_CHAR;
    ...
  end;
```

比如, 在 Linux 或在跨平台的情况下, 需要用以下方式处理消息:

```
const
  ID__REFRESH = $0001;
type
  TTextBox = class (TCustomControl)
  private
    procedure Refresh (var Message: TMessageRecordType); message ID__REFRESH;
    ...
  end;
```

message 方法不必包含 override 限定符来覆盖一个继承的 message 方法。实际上, 在覆盖方法时也不必指定相同的方法名称和参数类型, 而只要一个消息编号就决定了这个方法响应哪个消息和是否覆盖一个方法。

实现一个 message 方法时, 可以调用继承的 message 方法, 就像下面的例子:

```
procedure TTextBox.WMChar (var Message: TWMChar);
begin
  if Message.CharCode = Ord (#13) then
    ProcessEnter
  else
```

```
    inherited;  
end;
```

上面的例子是在 Windows 平台下使用的。Linux 或跨平台情况下可以按下面的例子实现相同的功能：

```
procedure TTextBox.Refresh (var Message: TMessageRecordType);  
begin  
    if Chr (Message.Code) = #13 then  
        ...  
    else  
        inherited;  
    end;  
end;
```

注意，示例程序中的 `inherited` 命令按照类的层次结构向后寻找，它将调用和当前方法具有相同消息编号的首个 `message` 方法，并把消息记录（参数）自动传给它。如果没有祖先类实现 `message` 方法来响应给定的消息编号，`inherited` 将调用 `TObject` 的 `DefaultHandler` 方法。

`DefaultHandler` 除了简单地返回外，不做任何事情。通过覆盖 `DefaultHandler` 方法，一个类可以实现自己对消息的响应。在 Windows 下，VCL 控件的 `DefaultHandler` 方法调用 Windows 的 `DefWindowProc` (API) 函数。

消息处理方法很少直接使用，相反，消息是通过继承自 `TObject` 的 `Dispatch` 方法来分派给对象的。

```
procedure Dispatch (var Message);
```

传给 `Dispatch` 的参数 `message` 必须是一个记录，并且它的第一个字段是 `Cardinal` 类型，用来存储消息编号。`Dispatch` 按照类的层次结构向后寻找（从调用对象所属的类开始），它将调用和传给它的消息具有相同编号的那个 `message` 方法。如果没有发现指定编号的 `message` 方法，则调用 `DefaultHandler`。

2.2.3 方法的绑定机制

所谓的方法绑定就是建立方法调用（Method Call）和方法本体（Method Body）之间的关联。如果绑定动作发生于程序执行之前（由编译器和连接器完成，简称编连），就称为“早绑定”或“先期绑定”（early binding）；如果绑定动作将在程序执行期才根据对象类型进行，则称为“晚绑定”或“后期绑定”（late binding）。晚绑定也被称为运行期绑定（run-time binding）或动态绑定（dynamic binding）。

在面向过程的编程中，既不可能用到也很少能听到“绑定”这个术语。因为面向过程的语言都是早绑定的，没有给程序员选择的机会。

Delphi 作为面向对象的语言可以实现不同的绑定机制，充分实现程序员的想像力。特别是功能强大的晚绑定机制，可以在运行期判定对象的类型，并调用其相应的方法。也就是说，编译器无需知道对象的类型，但方法的绑定和调用机制能够找出正确的方法体并加以调用。

根据方法的绑定机制，我们可以将方法划分为静态（static）方法、虚（virtual）方法和动态（dynamic）方法，其中虚方法和动态方法又可以是抽象（abstract）方法。

示例程序 2-4 演示了不同方法的声明格式。

示例程序 2-4 静态方法、虚方法和动态方法的声明

```
TMyClass = class
  procedure IAmAStatic; // 静态方法
  procedure IAmAVirtual; virtual; // 虚方法
  procedure IAmADynamic; dynamic; // 动态方法
  procedure IAmAVirtualAbstract; virtual; abstract; // 虚抽象方法
end;
```

1. 静态方法

在示例程序 2-4 中, IAmAStatic 是一个静态方法, 静态方法是方法的缺省类型, 对它就像对通常的过程和函数那样调用。当调用一个静态方法时, 类或对象所声明的类型决定了哪种实现(方法体)被执行。也就是说, 调用哪种方法实现是在编译时决定的, 编译器知道这些方法的地址, 所以调用一个静态方法时它能把运行信息静态地链接进可执行文件。静态方法执行的速度最快, 但它们却不能被覆盖来支持多态性。

在下面的例子中, Draw 方法是静态的。

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class (TFigure)
    procedure Draw;
  end;
```

按照上面的声明, 下面的代码演示了静态方法的执行结果。在第 2 个 TFigure.Draw 中, 变量 Figure 引用的是一个 TRectangle 类型的对象, 但却执行 TFigure 中的 Draw 方法, 因为变量 Figure 声明的类型是 TFigure。

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw; // 调用的是 TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw; // 调用的是 TFigure.Draw
  TRectangle (Figure).Draw; // 调用的是 TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw; // 调用的是 TRectangle.Draw
  Rectangle.Destroy;
end;
```

2. 虚方法和动态方法

要实现虚方法或动态方法, 在方法声明时要包含 virtual 或 dynamic 限定符。不同于静态方法, 虚方法和动态方法能在派生类中被覆盖。当调用一个被覆盖的方法时, 类或对象的实际类

型决定了哪种实现被调用，而不是它们被声明的类型。这一切都动态地发生在运行时，而不是在程序编译连接时。这意味着使用虚方法或动态方法在编程实现上有着极大的灵活性。

在示例程序 2-4 中，`IAmAVirtual` 是一个虚方法。虚方法和静态方法的调用方式相同。由于虚方法能被覆盖，在代码中调用一个指定的虚方法时编译器并不知道它的地址；因此，编译器通过建立虚方法表（VMT）来查找在运行时的函数地址。所有的虚方法在运行时通过 VMT 来调度，一个对象的 VMT 表中除了自己定义的虚方法外，还有其祖先的所有虚方法，因此虚方法比动态方法用的内存要多，但它执行得比较快。

在示例程序 2-4 中，`IAmADynamic` 是一个动态方法，动态方法与虚方法基本相似，只是它们的调度系统不同。编译器为每一个动态方法指定一个独一无二的编号，用这个编号和动态方法的地址构造一个动态方法表（DMT）。与 VMT 表不同，在 DMT 表中仅有它声明的动态方法，并且这个方法需要祖先的 DMT 表来访问它其余的动态方法。正因为这样，动态方法比虚方法用的内存要少，但执行起来较慢，因为有可能要到祖先对象的 DMT 中查找动态方法。

要覆盖一个方法，使用 `override` 限定符重新声明它即可。声明被覆盖的方法时，它的参数类型和顺序以及返回值（如果有的话）必须和祖先类相同。

在下面的例子中，`TFigure` 中声明的 `Draw` 方法在它的两个派生类中就被覆盖了。

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class (TFigure)
    procedure Draw; override;
  end;
  TEllipse = class (TFigure)
    procedure Draw; override;
  end;
```

给定上面的声明，下面的代码演示了虚方法在调用时的结果。大家可以注意到，在程序运行时，对于执行方法的变量而言，它的实际类型是变化的。

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw; // 调用的是 TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw; // 调用的是 TEllipse.Draw
  Figure.Destroy;
end;
```

3. 抽象方法

抽象方法是一种在声明它的类中暂时没有实现，而由它的派生类来实现的虚方法或动态方法。声明抽象方法时，必须在 `virtual` 或 `dynamic` 后面使用限定符。例如：

```
procedure DoSomething; virtual; abstract;
```

在示例程序 2-4 中，`IAmAVirtualAbstract` 是一个虚抽象方法。

只有当抽象方法在一个类中被覆盖时,才能使用这个类或它的实例进行调用。如果你创建基类的一个实例并强行调用它的一个抽象方法,Delphi 会调用 `AbstractErrorProc` 过程或者产生一个运行时错误 210 (`EAbstractError`)。

如果派生类没有实现(或者不需要实现)一个基类的抽象方法,则可以在类定义中忽略这个方法,或者用 `override` 和 `abstract` 限定符(必须是这个顺序)来声明该方法。后一种方案可以表明程序员的意图(表明该方法作为接口,在这个派生类中仍然是没实现的抽象方法),可避免错误调用没有实现的抽象方法。

4. 方法的覆盖、隐藏和重载

在 Delphi 中,通过覆盖一个方法可以实现 OOP 的多态性概念。通过覆盖可以使一方法在不同的派生类间表现出不同的行为。Delphi 中能被覆盖的方法是在声明时被限定为 `virtual` 或 `dynamic` 的方法。为了覆盖一个方法,在派生类的声明中用 `override` 代替 `virtual` 或 `dynamic`。例如,能用下面的代码覆盖 `IAmAVirtual` 和 `IAmADynamic` 方法:

```
TMyChildClass = class (TMyClass)
  procedure IAmAVirtualAbstract; override;
  procedure IAmADynamic; override;
end;
```

用了 `override` 关键字后,编译器就会用新的方法替换 VMT (虚方法表) 中原先的方法。如果用 `virtual` 或 `dynamic` 替换 `override` 重新声明 `IAmAVirtualAbstract` 和 `IAmADynamic`,就将是建立新的方法而不是对祖先的方法进行覆盖。同样,在派生类中如果企图对一个静态方法进行覆盖,则在新对象中的方法就完全替换在祖先类中的同名方法。

在下面的例子中, `TMan` 中声明的 `SayHello` 方法在它的两个派生类中被覆盖了。

```
type
  TMan = class (TObject)
  public
    function SayHello: string; virtual;
  end;
  TChinese = class (TMan)
  public
    function SayHello: string; override;
  end;
  TAmerican = class (TMan)
  public
    function SayHello: string; override;
  end;
```

给定上面的声明,下面的代码演示了虚方法被调用时的结果。在运行时,执行方法的变量的实际类型是变化的。

```
var
  AMan: TMan;
begin
  AMan := TChinese.create;
  edit1.text := AMan.SayHello; //调用的是 TChinese.SayHello
  AMan.free;
  AMan := TAmerican.create;
```



```

edit1.text: = AMan.SayHello; //调用的是 TAmerican.SayHello
AMan.free;
end;

```

在声明方法时，如果它和继承的方法具有相同的名称，但它不包含 `override` 限定符，则新方法仅仅是隐匿了继承下来的方法，并没有覆盖它。这两个方法在派生类中都存在，方法名是静态绑定的。比如：

```

type
  T1 = class (TObject)
    procedure Act; virtual;
  end;
  T2 = class (T1)
    procedure Act; //重新声明 Act,但没有覆盖
  end;
var
  SomeObject1: T1;
  SomeObject2: T2;
begin
  SomeObject1 := T2.Create;
  SomeObject1.Act; //调用的是 T1.Act
  SomeObject2 := T2.Create;
  SomeObject2.Act; //调用的是 T2.Act
end;

```

有时候，需要在派生类中增加一个方法，而这个方法的名称与祖先类中的某个方法名称相同。在这种情况下，没必要覆盖这个方法，只要在派生类中重新声明这个方法即可。但在编译时，编译器就会发出一个警告，告诉你派生类的方法将隐藏祖先类的同名方法。要解决这个问题，可以在派生类中使用 `reintroduce` 指示符，下面的代码演示了 `reintroduce` 指示符的正确用法：

```

type
  T1 = class (TObject)
    procedure Act; virtual;
  end;
  T2 = class (T1)
    //T2.Act 方法隐匿了 T1.Act 方法, reintroduce 限定符避免了编译器警告。
    procedure Act; reintroduce;
  end;

```

就像普通的过程和函数，方法也支持重载。一个方法可以使用 `overload` 限定符来重新声明，此时，若重新声明的方法和祖先类的方法具有不同的参数，则它只是重载了这个方法，而并没有隐匿它。当派生类中调用此方法时，依靠参数来决定到底调用哪一个。

如果要重载一个虚方法，应该在派生类中重新声明时使用 `reintroduce` 限定符。例如：

```

type
  T1 = class (TObject)
    procedure Test (I: Integer); overload; virtual;
  end;
  T2 = class (T1)
    procedure Test (S: string); reintroduce; overload;
  end;
...

```

```
SomeObject := T2.Create;  
SomeObject.Test ('Hello! '); //调用的是 T2.Test  
SomeObject.Test (7);        //调用的是 T1.Test
```

2.3 可见性

类成员包括数据成员和方法，类的每个成员都有一个称为可见性的属性，用来保护类成员。Delphi 有四种类成员的保护方式，分别为 `published`、`public`、`protected`、`private`。它们决定了一个类成员在哪些地方以及如何能被访问，`private` 表示最为隐秘的访问程度，`protected` 表示中等程度的访问能力，`published` 和 `public` 表示最大程度的访问能力。加上这些类的保护限定符，类定义的语法可改进为：

```
type 类名 = class (基类)  
    public  
    {类成员定义;}  
    protected  
    {类成员定义;}  
    private  
    {类成员定义;}  
    published  
    {类成员定义;}  
end;
```

在缺省情况下，类成员的可见性是 `public`（当在 `{ $M + }` 状态下编译类时，可见性是 `published`）。为了使程序具有良好易读的风格，建议最好在声明类时用可见性来组织类成员。

在继承关系中，可以在派生类中通过重新声明来增大一个类成员的可见性，但不能降低它的可见性。比如，一个 `protected` 属性在派生类中能被改变为 `public`，但不能改为 `private`。另外，`published` 成员在派生类中不能改为 `public`。

`private` 成员仅在该类的方法中被访问，它的派生类和实例都无法访问。但是 Delphi 的 `private` 并不是严格意义上的，如果将相关的类声明都放在一个模块中（即同一个单元文件中），这些类就可以访问同模块的其他类的私有成员。通过私有成员的限制，可以更好地封装和保护自己的类；清楚地向用户表明，他们无需关心这些与他们无关的项。

`protected` 成员在声明它的类的模块中是随处可用的，并且在它的派生类中也是可用的，无论该派生类出现在哪个模块。也就是说，在派生类的所有方法定义中，既可以调用 `protected` 方法，也能读取或写入 `protected` 数据成员或属性。只有在派生类的实现中才应用的成员通常使用 `protected` 属性。`protected` 成员和 `private` 成员的本质区别在于类的继承上，就是说，仍然可以通过继承在子类中访问 `protected` 成员，而 `private` 成员仅供自己的类使用。

`public` 成员是完全可访问的成员，可见性最大。虽然该成员使用方便，不受限制，但在编程中不能滥用。通常在设计中应该保持 `public` 成员的简明，并尽早定义使之稳定。因为 `public` 成员作为公共接口显然会在很多地方用到，设计不慎既会对使用该接口的其他类带来影响，又会威胁到其自身类的封装性。

`published` 成员和 `public` 成员具有相同的可见性，不同之处是 `published` 成员会产生 RTTI（Runtime Type Information 的简称，也有译成“运行类型库”、“运行期类型库”、“运行时刻类型

信息”的，本书考虑目前没有统一译法，所以直接使用 RTTI)。RTTI 使应用程序能动态查询一个对象的数据成员和属性，也能定位它的方法。RTTI 用于在存储文件和文件导入时访问属性的值，也用于在 Object Inspector 中显示属性，并且能为一些特定的属性（即事件）关联特定的方法（即事件处理程序）。

可公布属性的数据类型是受限制的。有序类型、字符串、类、接口和方法的指针可以被公布；当集合类型的基础类型是有序类型，且上界和下界介于 0 到 31 时，集合类型也是可以公布的。也就是说，集合必须符合 byte、word 或 double word。除了 Real48 外，任何实数类型都是可以公布的；数组类型的属性（区别于数组属性）不能是公布的。

一些属性虽然是可以公布的，但不能完全支持流系统，它们包括：记录类型的属性、所有可公布类型的数组属性以及包含匿名值的枚举类型的属性。如果 published 属性属于前面所述的类型，在 Object Inspector 中则不能正确显示这些属性，并且使用流向磁盘操作时也不能保存它们的值。

所有的方法都可以是公布的，但一个类不能使用相同的名字公布两个或两个以上的被重载的方法。只有当数据成员属于类或接口类型时，它才是可以公布的。

以上所有的种种限制是因为 published 声明指示编译器存储 VMT 中的信息，也就是说只有特定类型的信息可以存储。不过，如果你不需要让属性出现在 Object Inspector 中，则可以使用 public 限定符。至于将方法放在 published 声明部分是没有意义的，因为这与将其放在 public 声明部分效果相同。

2.4 属性

虽然 Delphi 是一种面向对象的编程语言，但许多抱有传统的面向过程编程思维的程序员仍然热衷于在 Delphi 中使用全局变量来访问数据。这种简单的思维方式往往导致数据莫名其妙地被修改或出现一些奇怪的值，给编程带来了极大的危害。另外，有一些使用过 C++ 的程序员，虽然知道要把数据封装在类中，使用 get 和 set 过程来读写数据；但这种方法使用繁琐，在简单求值时，无法直接使用过程作为操作数。

其实 Delphi 作为功能强大的面向对象开发利器，早就提供了一种通过过程访问数据的有效技术，但在使用上更像对数据的直接操作，这种“智能数据”就是属性。

2.4.1 什么是属性

属性就像是一个数据成员，它定义了对对象的一个特征。但数据成员仅仅是一个存储位置，它的内容可以被查看和修改，而属性通过读写它的值与特定的过程（函数）关联起来，保证了对这个值的读写是安全的、可以控制的。实际上，如果不使用过程管理数据，那么类中的数据与全局数据也没有什么区别。属性声明的语法如下：

```
property propertyName [indexes]: type index integerConstant specifiers;
```

其中：

- propertyName 属性名称可以是任何有效的标志符。

- [indexes] 是可选项，它是用分号隔开的参数声明序列，每个参数声明具有以下形式：
identifier₁, ..., identifier_n; type (详见 2.4.2 节)。
- type 必须是内置的或前面声明的数据类型，也就是说像 property Num: 0..9 ... 这样的属性声明是非法的。
- index integerConstant 子句是可选的。
- specifiers 是由 read、write、stored、default (或 no default) 和 implements 限定符组成的序列。每个属性声明必须至少包括一个 read 或 write 限定符。

属性由它们的访问限定符定义。与数据成员不同，属性不能作为变量参数向过程传递，也不能取得属性的地址。也就是说属性不能作为 var 参数传递，也不能使用 @ 运算符。这是因为属性无需存在于内存中。比如，它可能有一个读方法从数据库中检索数据值，或产生一个随机数据。

每个属性至少都有一个读限定符或一个写限定符，或两者都有。它们称为访问限定符。语法形式如下：

```
read fieldOrMethod  
write fieldOrMethod
```

其中，fieldOrMethod 是指一个数据成员或方法名，它们既可以和属性在同一个类中声明，也可以在祖先类中声明。

- 如果 fieldOrMethod 和属性是在同一个类中声明的，则它必须出现在属性声明的前面；如果它是在祖先类中声明的，则对派生类必须是可见的。也就是说，如果祖先类在不同的单元声明，则 fieldOrMethod 不能是私有的字段或方法。
- 若 fieldOrMethod 是一个数据成员，则它的类型和属性必须相同。
- 若 fieldOrMethod 是一个方法，则它不能是虚方法或动态方法，不能重载，而且对于公布的 (Published) 属性，访问方法必须使用默认的 register 调用约定。
- 在读限定符中，若 fieldOrMethod 是一个方法，则它必须是一个不带参数的函数，并且返回值和属性具有相同的类型。
- 在写限定符中，若 fieldOrMethod 是一个方法，则它必须是一个带有单一值参数或常量参数 (const) 的过程，这个参数和属性具有相同的类型。

例如：给定以下声明：

```
property Color: TColor read GetColor write SetColor;
```

则 GetColor 方法必须被声明为：

```
function GetColor: TColor;
```

SetColor 方法必须被声明为：

```
procedure SetColor (Value: TColor);  
procedure SetColor (const Value: TColor);
```

(当然，SetColor 的参数名不必非得是 Value。)

当在表达式中使用属性时，通过在读限定符中列出的数据成员或方法读取它的值；当在赋

值语句中使用属性时，通过写限定符中列出的数据成员或方法对它进行写入。

在下面的例子中，我们声明了一个叫做 TCompass 的类，它有一个公布的属性 Heading。Heading 的值通过 FHeading 数据成员读取，写入时使用了 SetHeading 过程。

```
type
  THeading = 0..359;
  TCompass = class (TControl)
  private
    FHeading: THeading;
    procedure SetHeading (Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
  ...
end;
```

给出上面的声明，语句

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

对应于

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading (135);
```

在 TCompass 类中，读取 Heading 属性时没有执行任何命令，只是取回存储在数据成员 FHeading 中的值。另一方面，给 Heading 属性赋值变成了对 SetHeading 方法的调用。它的操作将是把新值存储在数据成员 FHeading 中。当然还可以包括其他命令，比如，SetHeading 可能以如下方式实现：

```
procedure TCompass.SetHeading (Value: THeading);
begin
  if FHeading < > Value then
  begin
    FHeading := Value;
    Repaint; // 刷新用户界面来反映新值
  end;
end;
```

如果声明属性时只有读限定符，则它是只读属性；若只有写限定符，则它是只写属性。当给一个只读属性赋值，或在表达式中使用只写属性时都将产生错误。

2.4.2 使用数组属性

数组属性是被索引的属性，其索引方式与内建的数组相似。它们经常用来表示下面的一些事物：列表中的条目、容器控件中的子控件和位图中的像素等等。实际使用以 TList 和 TStrings 等类型更为常见。

声明数组属性时包含一个参数列表，它指定索引的名称和类型，比如：

```
property Objects [Index: Integer]: TObject read GetObject write SetObject;
property Pixels [X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values [const Name: string]: string read GetValue write SetValue;
```


除了使用中括号代替圆括号外，索引参数列表的格式和过程（或函数）的参数列表相同。数组只使用有序类型的索引，而数组属性的索引能使用任何类型。

对于数组属性，访问限定符必须使用方法（method）而不是数据成员（field）。读方法必须是一个函数，它的参数数目、类型以及顺序必须和索引中列出的一致，并且返回值和属性是同一类型；对于写方法，它必须是一个过程，该过程的参数列表必须和索引中列出的参数数目、类型以及顺序一致，另外还要有一个和属性具有相同类型的值参（传值）或常量参数。

例如前面的属性可能具有以下读写方法声明：

```
function GetObject (Index: Integer): TObject;  
function GetPixel (X, Y: Integer): TColor;  
function GetValue (const Name: string): string;  
procedure SetObject (Index: Integer; Value: TObject);  
procedure SetPixel (X, Y: Integer; Value: TColor);  
procedure SetValue (const Name, Value: string);
```

一个数组属性通过使用属性索引来进行访问。比如以下代码：

```
if Collection.Objects [0] = nil then Exit;  
Canvas.Pixels [10, 20] := clRed;  
Params.Values ['PATH'] := 'C: \BIN';
```

就相当于：

```
if Collection.GetObject (0) = nil then Exit;  
Canvas.SetPixel (10, 20, clRed);  
Params.SetValue ('PATH', 'C: \BIN');
```

定义数组属性时可以在后面使用 default 限定符，此时数组属性变成类的默认属性，例如：

```
type  
  TStringArray = class  
    public  
      property Strings [Index: Integer]: string ...; default;  
      ...  
    end;
```

如果一个类有默认属性，就能使用缩写 object [index] 来访问这个属性，它就相当于 object.property [index]。

比如，给定上面的声明，StringArray.Strings [7] 可以缩写为 StringArray [7]。一个类只能有一个默认属性，在派生类中改变或隐藏默认属性可能导致无法预知的行为，因为编译器总是静态绑定一个对象的默认属性。

2.5 异常

2.5.1 异常是一种特殊的对象

当出现错误或其他事件打断了程序的正常执行时，将引发一个异常（exception）。异常把控制权交给了一个异常处理程序（exception handler），这使得我们可以把错误处理和正常的程序逻辑隔离开来。因为异常是一种特殊的对象，所以，我们可以应用对象的继承关系把异常分层组

织,在不影响现有代码的情况下还能引入新的异常对象。异常能传递包括错误消息在内的一些信息,把它们从异常发生点带到被处理的地方。这就是说,我们能够在程序中捕捉到异常,并在特定的地方对异常进行处理。如果程序中使用了 SysUtils 单元,Delphi 可以将常见的运行错误转化为异常,并使它们被捕捉处理。这些错误包括:内存不足、被零除、一般保护性错误(GPF)等。这就避免了程序莫名其妙地终止。

在面向过程的编程中,程序员习惯使用函数的返回值来报告程序的错误,通常 0 表示成功,负数或非零整数表示错误,不同的整数值代表不同的错误类型。这种测试错误的代码导致程序中出现大量的条件判断语句,增加了编程和排错的工作量。

在面向对象的编程中,异常对象封装了各种类型的错误。通过异常处理程序,程序一旦出现错误,将引发对应的异常,即抛出对应的异常对象。然后程序执行代码从异常抛出的位置返回,从而保护后面的代码不被执行。如果在没有使用异常处理程序的情况下发生错误,将在程序的调用栈中进行回溯,直到遇到下一个外部的匹配的异常处理程序为止。实际上 Delphi 的 VCL 对象框架使得整个应用程序被包含在一个大的异常处理程序之中,并提供了现成的异常类。即使你的程序中没有任何异常处理代码,最后 Delphi 也能帮你捕捉到异常。

所以,Delphi 的异常处理机制通过异常类来引发异常,避免了程序中使用大量的复杂错误判断语句。

异常是一种对象,所以任何类的实例都可以作为异常对象。但通常我们习惯从 SysUtils 单元的 Exception 类来派生异常,这样通过对象的继承机制可以提高编写异常对象的效率。

异常类的声明和其他类一样,下面是声明一个计算错误异常类的例子:

```
type
  EMathError = class (Exception)
    ErrorCode: Integer;
    ErrorInfo: String;
  end;
```

通过异常类的继承,新派生出的异常使得对异常的捕获和处理变得更加精细。

既然异常是一个对象,那么在引发异常时就需要创建这个异常对象。所谓抛出异常就是用 raise 语句调用异常类的构造函数,如下所示:

```
raise EMathError.create;
```

与普通对象不同的是,创建的异常对象会在异常处理后自动销毁,千万不能手工销毁异常对象。

2.5.2 如何捕捉和处理异常

Delphi 有 try...except 和 try...finally 两个相关的语句用于处理异常。try...except 语句设置一个异常处理程序,这个程序在错误出现时获取控制权。而 try...finally 语句并不明确处理异常,但可以保证语句的 finally 部分的代码即使在一个异常引发时也能得到运行。

使用一个 try...except...end 语句可以捕获异常。try...except...end 语句的用法是:

```
try
  {异常保护的程序块}
```

```
except
  {一系列异常处理程序}
on {异常类} : do {处理程序}

else
  {其他未指明异常类的处理程序}
end
```

如果一段由 try...except 进行异常保护的程序块没有发生异常，异常处理程序将被忽略，程序执行 try...except...end 后面的代码。如果发生异常，则进入 except...end 中的异常处理程序。

若异常处理程序中能找到对应的异常，则控制权交给第一个匹配的处理程序。当处理程序中指定的异常类和实际发生的异常属于同类，或者是它的祖先类时，称为匹配。

如果没有找到相应的异常处理程序，程序控制就转到 else 子句。

如果以上条件都不满足，则向外层回溯，直到找到下一个 try...except 程序套。

在 on 子句中声明的变量包含有对异常对象的一个引用。当异常处理程序结束后，Delphi 自动释放这个对象。如果 Delphi 到了调用堆栈的尾部而没有找到一个匹配的异常处理程序，它将调用 ExceptProc。ExceptProc 实际上是一个指针变量，指向一个有两个参数的过程：异常对象以及异常发生的地址。

注意 SysUtils 单元为程序异常和运行时错误提供了额外的帮助。它定义了 ErrorProc 和 ExceptProc 过程。ErrorProc 把运行时错误转为异常，例如一个堆栈溢出错误转为 EstackOverflow。ExceptProc 例程将显示异常消息，然后暂停程序。在一个控制台应用程序中，异常消息被写到标准输出，且在 GUI 应用程序中，它显示在一个对话框之中。

下面是一个使用 try...except 来处理异常的例子：

```
function ComputeSomething:
begin
  try
    .....
  except
    on Err1: EDivByZero do
      showmessage ('零不能作除数! ');
    on Err2: EMathError do
      showmessage (Err2.ErrorInfo);
    else
      raise ;//重新引发这个异常
    end;
  end;
end;
```

对于发生异常后必须进行清除工作，但又不能进行全面处理的过程或函数，需要重新引发一个异常。在上面的例子中，在异常块中使用 raise 重新引发了正在处理的异常。将无法处理的异常交给该函数以外的任何异常处理程序去处理。

如果一时无法解决问题，也可以将异常记录下来或者写入 Windows NT 日志以便查询。

```
try
  .....
```

```
except
  On E: Exception do
    LogException (E);
end;
```

这里的 LogException 就是一个记录异常信息的过程。

使用 try...finally 可以在出现异常后保证仍然能够释放占用的资源（例如分配了的内存）。当代码引发一个异常时，Delphi 搜索调用堆栈来寻找 try 语句。当它找到一个 try...finally 时，将执行语句的 finally 部分的代码，然后继续搜索堆栈来寻找异常处理程序。如果一个 finally 块引发了一个异常，旧的异常对象就被释放，Delphi 将处理新的异常。

try...finally 语句最普遍的用法是释放对象和其他资源。所以我们经常会用到这样的创建对象的代码：

```
variable := typename.Create;
try
  .....
finally
  variable.Free;
end;
```

以及处理文件的代码：

```
reset (MyFile);
try
  ..... {处理文件}
finally
  closeFile (MyFile);
end;
```

第3章 理解对象

3.1 对象的本质

许多选择 Delphi 作为开发工具的朋友一开始都是被 Delphi 强大的 RAD (Rapid Application Development, 快速应用开发) 功能所吸引。不少人误以为 RAD 只适合传统的面向过程编程方法, 更多人习惯于将一个应用模型按照使用功能和行为过程进行分解, 再用控件进行搭建, 并构造繁多的函数和子程序。其实这些观点和方法都是对 RAD 开发工具的一种曲解。Delphi 强大易用的 VCL 本身就是建立在面向对象基础上的庞大体系, Delphi 就是一种面向对象编程语言, 支持类、继承、多态等所有面向对象的机制。如果不了解 Delphi 面向对象的机制, 不掌握 Delphi 面向对象的开发方法, 就很难解决一些复杂的应用, 完成大型的项目, 也无法真正体会到 Delphi RAD 的真谛。

了解对象的本质, 是面向对象开发的关键。有了“对象”这块“敲门砖”, 你将步入对象世界的“神秘宫殿”, 领略面向对象开发的无限风光。

3.1.1 什么是对象

对象是客观世界中的事物在人脑中的映像, 这种映像通过对同一类对象的抽象反映成人的意识, 并做为一种概念面存在。我们知道, 客观世界是由许多不同种类的对象构成的, 每一个对象都有自己的运动规律和内部状态, 不同对象之间相互联系、相互作用。

计算机是人类用于认识世界和改造世界的工具, 作为计算机本身并不能识别和抽象作为客观世界反映的对象。我们通过编程, 实现计算机解决客观世界问题的桥梁。也就是说, 是程序员 (更广泛地说应该是开发人员) 建立了介于机器模型和客观世界模型之间的关联。在建立这种关联时, 由于编程语言的进化, 使得我们现在能够将编程 (开发) 的中心更多地向客观世界模型的一端迁移, 而不是过多地局限于机器模型。这样才使得我们能够在这里讨论 OOP 语言中的对象, 而不是汇编语言中的二进制代码。

面向对象技术是一种从组织结构上模拟客观世界的方法, 从组成客观世界的对象着眼, 通过抽象, 将对象映射到计算机系统中, 又通过模拟对象之间的相互作用、相互联系来模拟现实客观世界, 描述客观世界的运动规律。所以, 在这里我们关心和讨论的是 Delphi 编程中的对象。

在 Delphi 编程中, 我们使用对象类别 (类) 或对象类型 (抽象类或对象接口) 来描述客观世界。这是因为类或类型可以抽象出一般规律和本质特征, 并可以根据客观世界的复杂性进一步继承生成新类, 还可以适应客观世界的变化性来实现多态。设计优秀的对象类别或对象类型体系, 可以做到以不变应万变, 实现软件的复用, 延长产品的寿命。对象作为类的实例有着自己的生命周期, 我们可以将客观世界的行为模型化为消息序列, 并在不同的对象之间进行传递, 实际上, 对象之间的通信是通过对象自身的行为实现的。通过这种途径, 我们能够以面向

对象的方式更容易地对软件进行设计和编程，以获得灵活性和可维护性。

编程中的对象与客观世界中的对象并不是完全相同的概念，它们既有联系又有区别。所以，深刻理解编程中的对象本质将有利于实践面向对象编程和开发。由于大多数开发者已经学习并使用了传统的面向过程编程方法，因此，形成了结构分析和功能分解的思维方式，导致了他们在编程中注重过程化的一步一步的操作。这种思维方式对客观世界的反映不太自然和直接，不利于解决关系复杂的问题。因此通过理解对象的本质，将有利于彻底地改变这种思维方式，以适应面向对象的编程。下面我们就从几个方面来理解 OOP 中对象的本质。

首先，对象可以视为一组相关的代码和数据的组合。它是包含相互之间有联系的过程集和数据的软件包。在面向对象程序设计中，过程（函数）被称做方法，数据被称做属性。对象的概念是简单的，但它的功能是强大的且灵活的。对象是个理性的软件模型，因为可以定义并保证它们相互之间独立，每一个对象都可形成一个自我包容的独立单元。可以用属性来表示对象的内容，用方法来表示对象的操作。

对象可以视为神奇的变量。它是“类”类型的变量，有自己的属性，可以存储数据；它有自己的行为，可以执行自身具备的操作。理论上，你可以把所有要解决的问题分解成程序中的各个对象，由它们自己解决各自的问题。

对象可以视为元数据。对象封装了方法和属性，并提供外部调用的接口。这使得对象可以作为一个独立的整体单元安全使用，其维护了自身的完整性和可操作性（见图 3-1）。当然，对象的粒度划分仍然是面向对象编程中的难题。

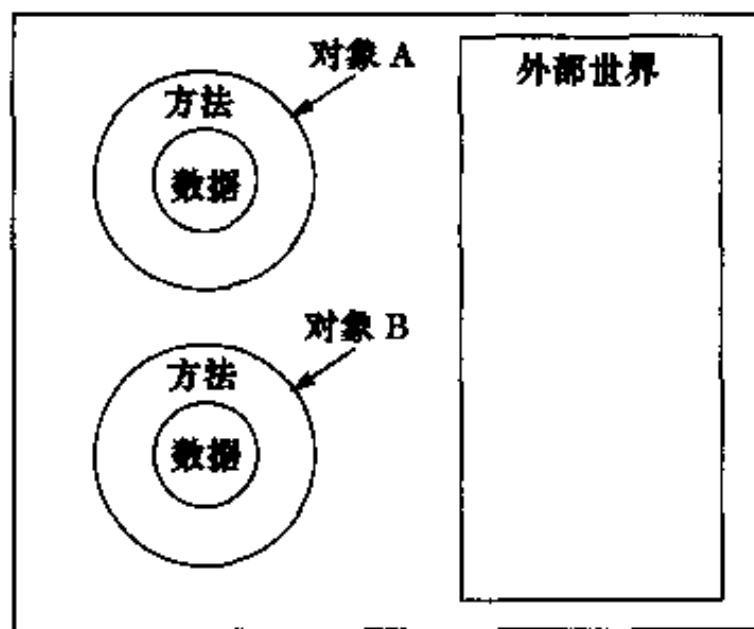


图 3-1 对象是一个整体，封装了对外部世界隐藏的数据

其次，对象可以互相协作，共同完成任务。对象之间可以通过发送消息请求而相互联系，在消息请求中可以调用方法和设置属性。消息由对象的名字后跟它的成员（方法或属性）来表示。一个消息通常由三部分组成：接收对象的名字、对象成员的名字（方法或属性）和对象成员参数。

还有，每个对象都有类型（Type），对象是类的实例。同一类型的所有对象可以接受相同的消息。比如：“中国人”对象也是“人”对象（TChinese 继承自 TMan），所以“中国人”能够接受所有发给“人”的消息。这意味着，我们只要写一次相关的消息程序，就可以自动处理同

一类型的所有对象。

另外,通过继承、组合或封装等方式可以生成新的对象,并以此在程序中构建复杂的体系,将系统的复杂性隐匿于对象的简易性之中。

最后,对象根据特定的意义和用途有不同的划分方法。你既可以划分账单对象、收银员对象这样的实体对象,也可以划分安全对象、协调对象、事务对象这样的功能对象。通常我们可以按照界面和逻辑分开的原则,将系统划分为系统逻辑对象和用户界面对象。

系统逻辑对象并不是一个可视化的对象,它封装了应用程序的系统逻辑。一个非可视化的对象是一个不包括可视化的控件和接口的对象。系统逻辑对象处理客户前端和数据后端的通信。它们提供了一个抽象的中间层,在这里可以变更客户前端或数据后端(或者同时变更客户前端和数据后端),但是系统逻辑保持不变。另外,程序的发行和维护变得简单了。如果系统逻辑因请求而发生变化,这种变化仅发生在一个位置,因此并不影响客户前端和数据后端。

系统逻辑对象提供这种服务:当客户机需要服务时,该相应的系统对象就会随之响应。当客户机需要与数据相连时,连接系统对象就会建立和提供这种连接。客户机提出 SQL 请求,SQL 系统对象则建立该请求和数据库服务器之间的通信,然后,数据库服务器处理这种请求,并通过 SQL 系统对象返回结果。客户机要响应一个缓冲计划,缓冲计划系统对象就会产生并返回这样的计划。

系统对象可以是 Java Beans、COM+ 组件、.NET 组件和 C++ 对象等等。Delphi 可以很容易地创建 COM+ 组件、VCL 组件和 DLL 等等。

用户界面对象,一般来说是一些可视化的对象,它封装了用户界面应用程序。用户界面的对象的例子包括日历、网格、选取框等等。在 Delphi 中,用户界面对象使用了 VCL 组件和功能强大的第三方控件,以实现图形窗体中丰富多彩的界面。

3.1.2 对象在哪里

我们知道,从语义上讲,对象是类的实例,类是创建对象的模板;从语言上讲,对象是类这种数据类型的变量,对象在内存中占有空间。但是在具体使用中,对象与传统的变量有什么区别呢?对象存储在哪里,我们如何调用和传递对象?

首先我们要搞清 Delphi 的“引用/值”模型。

在 Delphi 中,简单的数据类型(如: Integer、Char、Record 等)无论是作为参数还是变量都是按值传递和使用的,通常称之为值类型。值类型也是直接类型,它意味着当我们更改这个变量时,实际上是直接更改了它的数值。例如在示例程序 3-1 中,变量 j 得到的是变量 i 的一个副本,当变量 i 再次改变时,不会影响到变量 j。所以,运行程序,点击“值类型”按钮,显示了变量 i 值为 25,变量 j 值为 24,如图 3-2 所示。

而 Delphi 中复杂的数据类型(如: class)则是按引用传递和使用的。引用类型是间接类型,它与值类型存储数据的方式不同,它存储的是间接数据,即对该数据的引用。例如在示例程序 3-1 中,我们创建了一个称为 TMyClass 的类,变量 a 和 b 都是该类的实例,它们并不存储该对象的数据(注意,对象的数据保存在其数据成员中,如本例的 myvar。),它们只保存了该

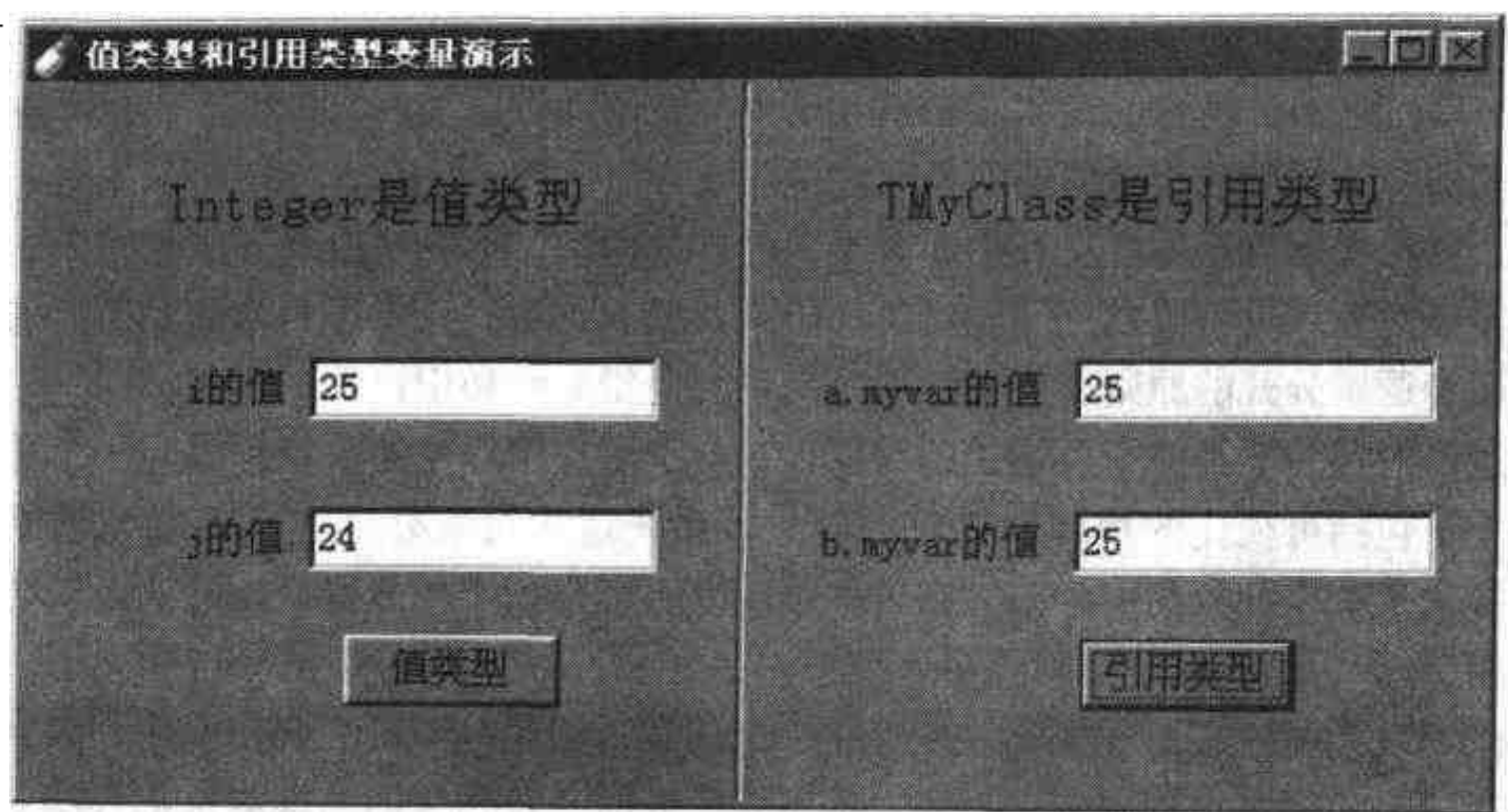


图 3-2 值类型和引用类型变量演示运行结果

对象的一个引用。这就是说，当变量 a 赋值给 b 时，实际上仅仅将对象的引用赋值给了 b。所以，当对象的数据（myvar）改变时，a 和 b 通过相同的引用，得到的是同一对象的数据成员值。如图 3-2 所示，我们看到这个值都是 25。

示例程序 3-1 值类型和引用类型变量演示

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class (TForm)
    edtByVar2: TEdit;
    btnByVar: TButton;
    edtByRef1: TEdit;
    edtByRef2: TEdit;
    btnByRef: TButton;
    edtByVar1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Bevel1: TBevel;
    procedure btnByVarClick (Sender: TObject);
    procedure btnByRefClick (Sender: TObject);
  private
    | Private declarations |
  end;
end;
```

```
public
  | Public declarations |
end;

TMyClass = class
  myvar: Integer;
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.btnByVarClick (Sender: TObject);
var i, j: integer;
begin
  i: = 24;
  j: = i; //变量 j 得到的是变量 i 的一个副本
  i: = i + 1;
  edtByVar1.Text: = inttostr (i);
  edtByVar2.Text: = inttostr (j);
end;

procedure TForm1.btnByRefClick (Sender: TObject);
var a, b: TMyClass;
begin
  a: = TMyClass.Create;
  a.myvar: = 24;
  b: = a; //变量 a 得到的是变量 b 的一个引用的副本
  a.myvar: = a.myvar + 1;
  edtByRef1.Text: = inttostr (a.myvar);
  edtByRef2.Text: = inttostr (b.myvar);
end;

end.
```

现在我们知道了值类型和引用类型变量有着很大的不同。当这两种变量（或参数）传递时，前者传递的是值的副本，后值传递的是引用的副本。为什么要这样呢？这是因为它们存储的地方不同，使用的内存机制不同。

计算机内存通常会按照功能和性能的不同划分成一些块，其中常用的有两块，称为栈（stack）和堆（heap）。在栈中，处理器直接使用栈指针（stack pointer）分配和访问内存。这种方式速度快、效率高，但必须由 Delphi 编译器来编译产生控制栈指针的程序代码，掌握数据在栈中占用的空间大小和存活时间。这样一来就限制了程序的灵活性，比如 Delphi 对基本数据类型的规定就比较死，编译时还要进行强制性检查。所以，我们不能将对象存储于栈中，而只能将对象的引用存储于栈中。从某种程度上讲，这也是考虑到对象的大小和生命期是不确定的，而对象引用的大小和生命期是可以确定的。因此，在栈中的变量是不需要由程序员手工去释放内存空间的。

堆是一种通用性质的内存存储空间，是真正用于存放对象的地方。堆和栈的不同之处在于，编译器无需知道对象究竟要从堆中分配多少内存空间，占用多长内存时间。因此，从堆中分配存储空间可以获得最大的灵活性，但是这种分配内存的方式需要手工进行，在 Delphi 中使用 `create` 方法或构造函数来实现对象的空间分配。同样，已分配的空间也要由程序员手工销毁，通常使用的是 `free` 方法。

从图 3-3 中，我们可以更形象地了解到简单数据类型的值和对象的引用存储在栈中，而对象则存储在堆中。我们在程序中通过对象的引用来访问该对象，实际上改变对象的引用并不能改变对象本身。

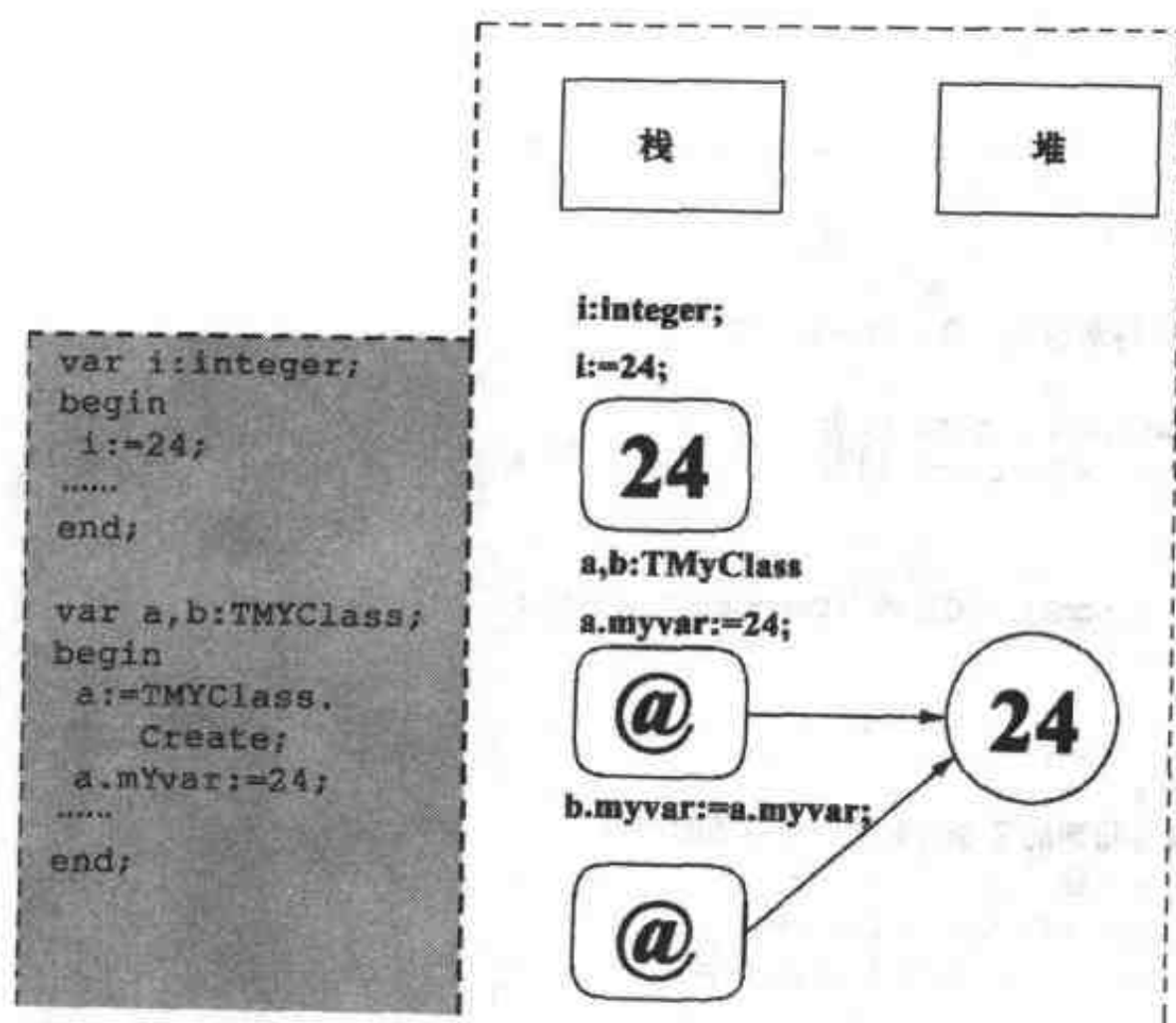


图 3-3 简单数据类型的值和对象的引用存储在栈中，对象存储在堆中

为了“享用”堆的好处，我们也要付出一定的代价。首先，使用堆在速度和效率方面要比栈差一点。其次，堆的手工分配和释放内存空间要比栈的自动管理内存空间麻烦一点。如果粗心的程序员在程序中忘记销毁对象，就可能造成内存的泄漏，影响计算机的整体工作效率。但是，大多数情况下并不是程序员忘记销毁对象，而是他们根本搞不清值和引用的关系，误以为对象变量不存在了，对象就不存在了。实际上有时作为变量的对象引用销毁了，并不代表该对象已经销毁。对象的引用就好像是对象的一个“手柄”，我们通过它来操纵对象。这就像我们使用一个有柄的锅，如果柄没断，我们可以通过柄扔掉这个锅，如果柄断了，我们扔掉的是柄而不是这个锅，如果没有柄我们可能无法扔掉锅。所以，当我们编程时千万不能把“柄”扔掉再去找“锅”，这样越积越多的无头对象肯定会塞满堆中，造成内存泄漏。

如果读者对自己创建的对象所占用内存的大小感兴趣，不妨调用该对象的 `InstanceSize()` 方法进行查看，所有对象都从 `TObject` 继承了 `InstanceSize()` 方法。

3.1.3 对象引用和类引用

前面,我们已经了解到,对象是类的动态实例,对象总是被动态分配到堆上。因此一个对象引用就如同一个句柄或一个指针。但你分配一个对象引用给一个变量时,Delphi 仅复制引用,而不是整个对象。当你的程序不再使用一个对象时,应当调用 Free 方法显式地销毁该对象。Delphi 没有提供自动的垃圾回收机制。

为简短起见,大家通常将“对象引用”简称为“对象”,但是我们在使用时务必有清醒的认识。在 Delphi 中,使用一个对象的惟一方法就是使用对象引用。一个对象引用通常以一个变量的形式存在,但是也有函数或者属性返回值的形式。

Ray Lischner 在《Delphi in a Nutshell》一书中认为 Delphi 中的类是一个独立的实体。在 Delphi 中,类表现为内存中一张只读的表,表中存放着指向该类的虚方法的指针以及其他许多信息。一个类引用(class reference)就是指向该表的一个指针。

示例程序 3-2 和图 3-4 说明了对象引用和类引用的不同。

示例程序 3-2 类和对象

```
type
  TAccount = class
  private
    fCustomer: string; // 客户姓名
    fNumber: Cardinal; // 账户号码
    fBalance: Currency; // 账户结余
  end;
  TSavingsAccount = class (TAccount)
  private
    fInterestRate: Integer; // 年利率
  end;
  TCheckingAccount = class (TAccount)
  private
    fReturnChecks: Boolean;
  end;
  TCertificateOfDeposit = class (TSavingsAccount)
  private
    fTerm: Cardinal; //存款单 (Certificate Of Deposit,简称 CD) 存期
  end;

var
  CD1, CD2: TAccount;
begin
  CD1 := TCertificateOfDeposit.Create;
  CD2 := TCertificateOfDeposit.Create;
  ...
```

从图 3-4 中可以看到, CD1 和 CD2 分别通过对象引用可以访问各自创建的对象。而它们的对象都是通过 TCertificateOfDeposit 类创建的,是 TCertificateOfDeposit 类的实例。虽然两个对象各自的数据成员值不一样(对象的状态不一样),但拥有的方法都一样(对象的行为一样)。这就

是说，同一个类创建的对象，通过类引用都指向了同一个类表。显然，在类的继承关系中，我们也能看到这种类引用，即派生类调用基类的方法时，它们指向了同一个类表。

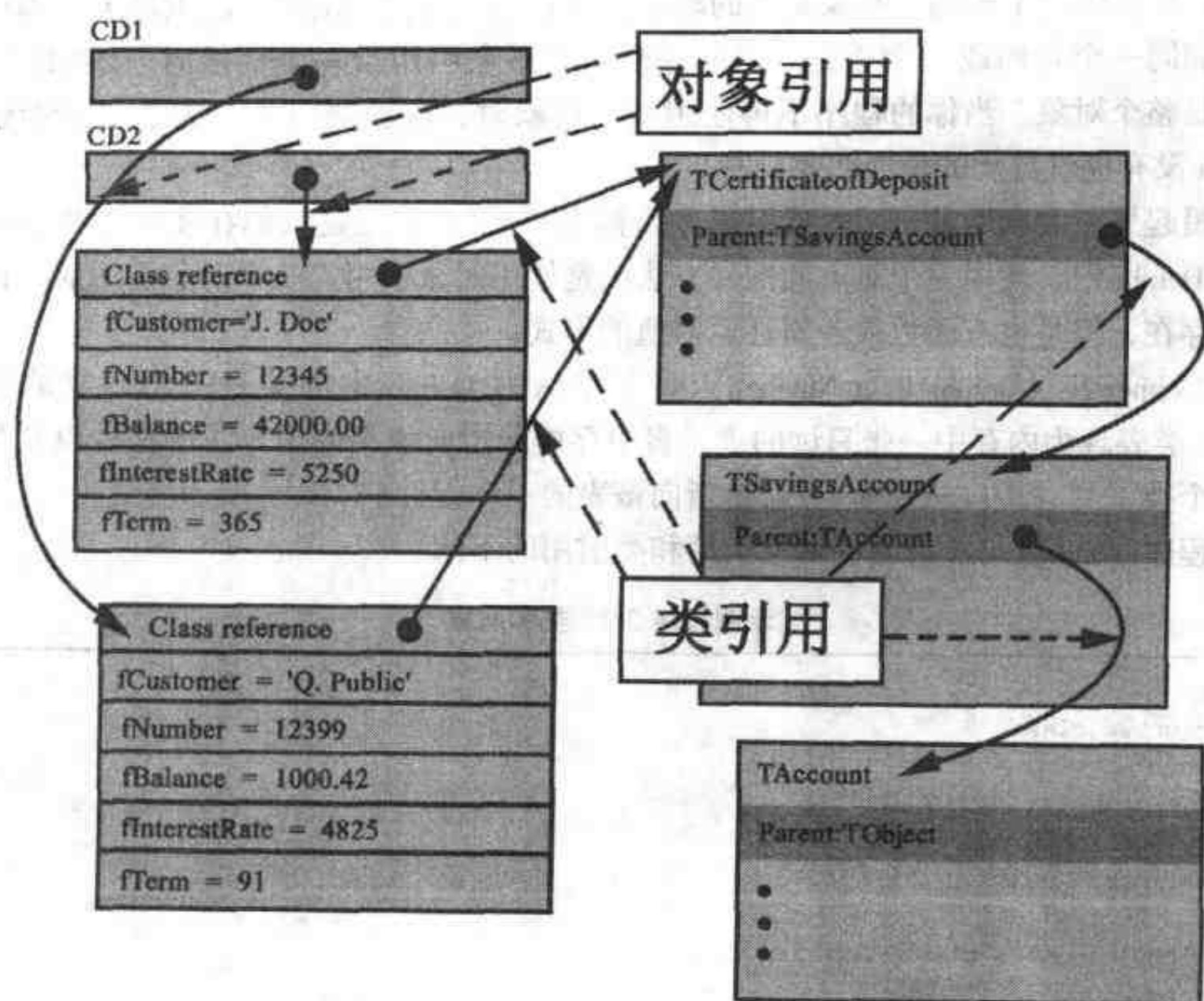


图 3-4 对象和类在内存中的布局

既然引用就如同一个句柄或一个指针，那么我们能不能像通过对象引用来操纵对象一样通过类引用来直接操纵类呢？答案是肯定的，实际上我们有时需要使用类本身而不是它的实例（对象），比如，当使用类引用来调用类的构造函数时，我们希望总能使用类名来引用一个类。还有，我们有时需要声明变量或参数，并把类作为它的值，在这种情况下，我们还要用到类引用类型。

所谓类引用类型（class-reference type）是一种“类的类”（class of class）类型，也称做元类（metaclass）。其构造形式为：class of *type*。这里，*type* 是任何类型。例如：

```
type TClass = class of TObject;
var AnyObj: TClass;
```

声明了一个叫做 AnyObj 的变量，它能存储任何类引用。类引用类型的声明不能直接用于变量或参数声明中。可以把 nil 赋给任何类引用变量。

类引用最常见的用法是创建一个对象或者用来测试一个对象引用的类型；也可以在其他许多场合使用。比如，传递类引用给某个例程或者从一个函数中返回一个类引用。

将类引用作为参数传递的好处在于，我们有时候无法确定某个对象的具体类型（但总是能知道这个对象的祖先类类型，大不了就是 `TObject`），若将类引用作为参数传递，则即使编译时也能通过类型检查，而实际上对象类型的确定是在程序运行时而不是在编译时，这样就增加了编程上的灵活性。

简单地说，如果把类看做是对象的模板，那么就可以把类引用类型（类的类）看做是类的模板。下面我来举一个例子说明。

图 3-5 是一个类引用演示程序的运行画面，通过这个例子，我们可以选择不同类型的控件，动态地在窗体中生成。

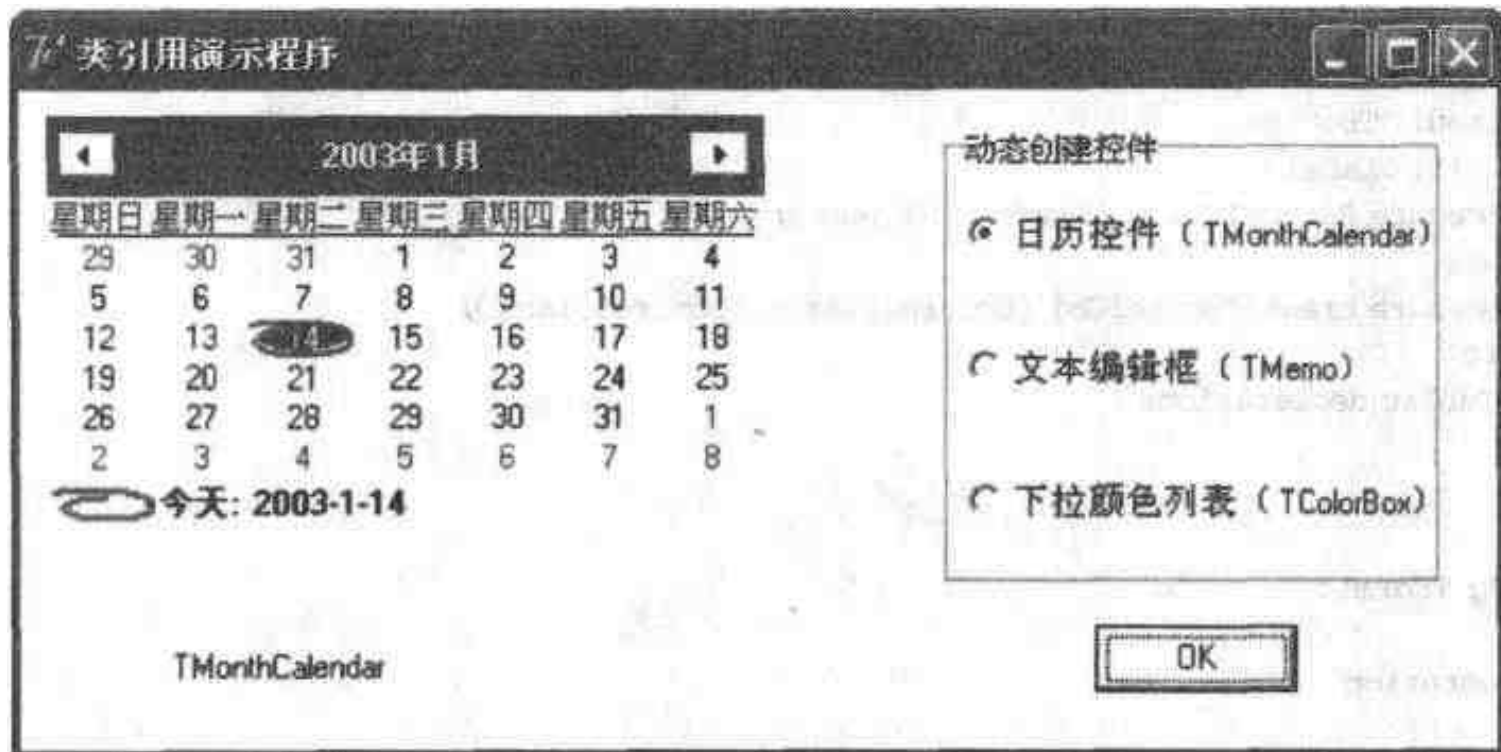


图 3-5 类引用演示程序动态创建控件

示例程序 3-3 是这个例子的源代码。这里并没有为每个类型的控件都分别写一个构造函数，而是用下面这个方法来的动态创建控件：

```
createControlObj (ControlClass: TControlClass);
```

在图 3-5 中，提供给用户 `TMonthCalendar`、`TMemo` 和 `TColorBox` 三种类型的控件选择。由于我们不知道用户会选择哪类控件（尽管可以用条件判断语句，但这不是好的编程方法，因为实际应用中问题更复杂），因此必须创建一个控件的类引用类型：

```
TControlClass = class of TControl;
```

`TControl` 是所有控件的祖先类，我们不必担心编译器的类型检查。如果是传递类型确定的对象，我们可能会这样写：

```
createControlObj (MyControl: TMonthCalendar);
```

但考虑到控件类型的多样性，我们现在是将控件的类引用类型作为参数传递给 `createControlObj` 方法，这样无论用户选择哪类控件，都可以通过类引用找到该控件的类，并创建对应类的实例（对象）。一言以蔽之，我们作为参数传递的是类引用而不是对象引用。所以说，这种思维方式的巧妙之处在于构造函数可以通过一个类引用类型的变量进行调用，这就可以创建编译时类型不确定的对象。

示例程序 3-3 类引用演示程序

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, ComCtrls;

type
  TControlClass = class of TControl; //声明类引用类型
  TForm1 = class (TForm)
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click (Sender: TObject);
  private
    procedure createControlObj (ControlClass: TControlClass);
  public
    | Public declarations |
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
procedure TForm1.createControlObj (ControlClass: TControlClass);
  var AControlObj: TControl;
begin
  AControlObj := ControlClass.Create (self); //根据不同的类类型,动态创建对象
  AControlObj.Parent := self; //指定 TForm1 为控件的容器
  AControlObj.Name := AControlObj.ClassName; //以类名作为控件名
  AControlObj.SetBounds (10, 10, 250, 150); //设置控件边界
  Label1.Caption := AControlObj.ClassName;
end;

procedure TForm1.Button1Click (Sender: TObject);
const
  //定义一个以控件类类型为元素的数组
  CtrlClassArray: array [0..2] of
    TControlClass = (TMonthCalendar, TMemo, TColorBox);
var
  i: integer;
begin
  //清空已经动态生成的控件
  for i := 0 to controlcount-1 do
    if (Controls [i] is TMonthCalendar) or (Controls [i] is TMemo)
      or (Controls [i] is TColorBox) then
      Controls [i].Free;
  //调用构造函数,动态生成用户选定的控件
  createControlObj (CtrlClassArray [RadioGroup1.ItemIndex]);

```

```
end;
```

```
end.
```

通过上面的例子，我们进一步认识到类引用是引用一个具体类的表达式。类引用在 Delphi 中用做生成新对象、调用类方法，以及测试或者转换对象类型。类引用实现为指向一个关于该类的信息表的指针，特别是类的虚方法表（VMT）。

类引用最普遍的使用是通过调用构造函数来生成该类的实例。也可以用类引用来测试对象类型（用类操作符 `is`）或者将对象转换为特定类型（用类操作符 `as`）。通常，类引用的函数是一个类的名称，但也可以是类型为元类的变量，或者返回为类引用的函数或属性。

3.1.4 对象的传递

说到这里，我们不妨看一下下面两段经常读到的程序：

```
procedure SetEdit (Edit: TEdit);  
begin  
  Edit.Text: = 'ABC';  
end;
```

和

```
procedure SetEdit (var Edit: TEdit);  
begin  
  Edit.Text: = 'ABC';  
end;
```

这两种写法有什么差别？效果会不会一样？要搞清这个问题首先得了解 Delphi 的参数传递机制。

在 Delphi 中，我们常用的参数传递机制分为值传递和引用传递，前者传递的是数值参数（默认），后者传递的是变量参数（`var`）。要理解这句话的意义，可考虑以下函数：

```
function PlusByValue (x: integer): integer; //x是数值参数,传递值  
begin  
  x: = x + 1;  
  result: = x;  
end;  
  
function PlusByRef (var x: integer): integer; //x是变量参数,传递引用  
begin  
  x: = x + 1;  
  result: = x;  
end;
```

虽然这两个函数返回同样的结果，但第二个函数（`PlusByRef`）能改变传给它的变量的值。假设我们调用下面函数：

```
var I, J, V, W: integer;  
begin  
  I: = 4;
```

```
V := 4;  
J := PlusByValue (I); //J = 5, I = 4  
W := PlusByRef (V); //W = 5, V = 5  
end;
```

程序执行后，I 传递给 PlusByValue 函数，在 PlusByValue 产生的 x 是 I 的一个副本，数值参数此时就好像是一个局部变量，其变化并不影响 I 的值；不同的是，V 传递给 PlusByRef 函数，传递的是 V 的引用，即 x 是指向 V 变量值的指针，变量参数在函数中的变化，直接影响到 V 的值。使用引用传递机制时，要注意变量参数即使传递给多个参数，也不会创建它的副本，感觉上变量参数名本身好像是超出了它的作用域。例如：

```
procedure PlusAndTimes (var x, y: integer); integer;  
begin  
  x := x + 5;  
  y := y * 5;  
end;  
  
var I: integer;  
begin  
  I := 5;  
  PlusAndTimes (I, I);  
end;
```

运行程序，I 值结果是 50。显然，这里不能再将参数 x、y 看成是 PlusAndTimes 过程中的局部变量。

那么，对象在作为参数的传递中是否也遵循这一原则呢？当然也遵循。不过这里要注意的是对象变量本身就是引用类型，我们在传递对象时，实际上传递的都是对象的引用。所以一开始我们看到的 SetEdit 过程的两种写法虽然不同，但效果是一样的。前一种写法是典型的值传递机制，它把一个 TEdit 类型的对象引用在 SetEdit 过程中复制了一份。值得注意的是，复制得到的是对象引用的副本，而不是对象的副本。该对象引用的副本类似于一个局部变量，它的生命期仅限于 SetEdit 过程中。该过程一旦结束，它就被抛弃了，但 TEdit 的对象还在，该对象原来的引用还在。第二种写法是把一个 TEdit 对象引用直接作为参数传递，可以认为传递的是该对象的指针。

由此可见，由于对象变量本身就是对象的引用，所以当对象作为参数传递时，无论是采用值传递机制还是引用传递机制，传递的都是对象的引用。所以这两种写法的效果实际上是一样的。不同的是，第一种写法在 SetEdit 过程中产生了对象变量（引用）的另一个副本。

对象变量的副本可以视做是该对象的别名，如果一个对象有多个别名，这就意味着多个引用绑定到同一个对象。如果有人通过某个别名改动了这个对象，而其他一些对象别名的使用者不知道这种改变，这就会导致他们无法理解的结果，引发问题。（即使是你一个人编程，也有可能被对象的多个别名搞糊涂，在不甚明白的情况下改动了对象。）

要解决这个问题，首先你要明确对象和对象引用的关系，理解对象传递的实质，这也是我在这里详细讲解的内容；其次，尽量少用对象的别名，不要在同一范围内（比如一个过程和函数中）产生一个以上的对象别名，这样即使有了对象的别名，它们也会限制在各自的生命期中

(也可以理解成该变量使用的范围中);最后,如果你的确想保护你的对象不受改动,还可以考虑制作一个该对象的真正副本,也就是说克隆一个对象。

3.1.5 对象的克隆

在 VCL 的体系结构中,TPersistent 类系下的对象都是可以提供克隆行为的。TPersistent 类是抽象类,没有实例对象。但是 TPersistent 类提供了一个接口,引入了对象的可赋值性和流化 (assignment and streaming capabilities) 等性质。这里,我们要搞清楚进行对象赋值时的两个不同的概念,其一,是使用赋值操作符 (`:` `=`) 将一个对象的引用赋值给一个对象变量;其二,是使用 Assign 或 AssignTo 方法可以将对象属性进行复制,得到两个状态一样的对象。

进行第一种赋值操作时,并不需要对象的实例,只需要对象变量的引用即可:

```
var
  a, b: TMyObject;
begin
  a := TMyObject.create;
  b := a;
end;
```

如果写成下面这样,反而会导致内存泄漏:

```
var
  a, b: TMyObject;
begin
  a := TMyObject.create;
  b := TMyObject.create;
  b := a; //错误,对象 b 丢失导致内存泄漏。
end;
```

如果要想对象 b 克隆对象 a,则需要考虑第二种赋值操作:

```
var
  a, b: TMyObject; //假设这里的 TMyObject 是 TPersistent 的派生类
begin
  a := TMyObject.create;
  { 关于对象 a 的代码 }
  .....
  { 开始克隆对象 a }
  b := TMyObject.create;
  b.Assign(a); //对象 b 的属性和内容和对象 a 完全相同。
end;
```

由此可见, `b := a` 意味着 b 是 a 的引用,即两者是同一对象。如果写成 `b.Assign(a)`,那么 b 是一个单独的对象,其状态与 a 相同,也就可以看成是 b 克隆了 a。

为了提高代码的效率和质量,在编程中经常需要活用对象克隆技术。我们来看一个简单的例子。一位程序员需要设计一个可以在文本编辑框中 (TMemo) 任意设置字体,并可恢复到原字体的程序,如图 3-6 所示。他写了下面一段程序 (示例程序 3-4),发现可以设置字体,但无法复原。

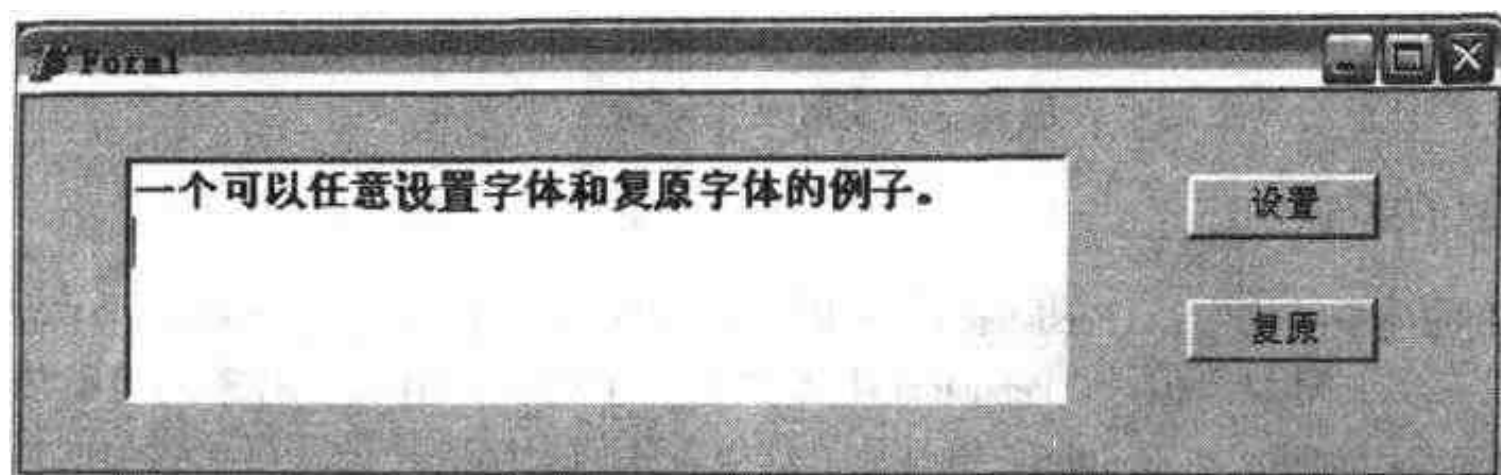


图 3-6 一个可以任意设置和复原字体的例子

示例程序 3-4 首次设计的程序无法复原字体

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TForm1 = class (TForm)
    FontDialog1: TFontDialog;
    btnUndo: TButton;
    btnSet: TButton;
    Memo1: TMemo;
    procedure btnSetClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
    procedure btnUndoClick (Sender: TObject);
  private
    FOriginalFont: TFont;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.btnSetClick (Sender: TObject);
begin
  if FontDialog1.Execute then
    Memo1.Font := FontDialog1.Font;
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
  Memo1.Lines.Add (

```



```
'一个可以任意设置字体和复原字体的例子。');  
FOriginalFont: = Memo1.Font;  
end;  
  
procedure TForm1.btnUndoClick (Sender: TObject);  
begin  
    Memo1.Font: = FOriginalFont;  
    //Memo1.Font 和 FOriginalFont 实际上引用的是同一个对象  
end;  
  
end.
```

我们分析示例程序 3-4, 可以发现这位程序员的问题在于把对象变量和普通的值类型变量混淆了。在 Delphi 中, 简单的数据类型 (如: Integer、Char、Record 等) 无论是作为参数还是变量都是按值传递和使用的, 通常称之为值类型, 而对象变量是引用类型变量, 但你分配一个对象引用给一个变量时, Delphi 仅复制引用, 而不是整个对象。因此这位程序员虽然在 FormCreate 事件中写下了 “FOriginalFont: = Memo1.Font;” 代码以期保存原来的字体, 但实际上他只是复制了 Memo1.Font 的对象引用, 也就是说 Memo1.Font 和 FOriginalFont 引用的是同一个对象, 该对象后来变成了 FontDialog1.Font。所以, 当然无法从 FOriginalFont 还原字体了。

有意思的是, 后来这个程序员把代码改成示例程序 3-5 所示的那样, 发觉可以实现字体复原功能了。其实这样的程序改动虽然解决了一些问题, 但仍然不是最好的办法。试想, 一个 TFont 对象不仅仅包含名称、大小、字形的属性, 它还有颜色等许多属性。如果这样修改程序, 需要考虑很全面才行。另外, 逐一地保存和恢复这些属性状态的确也很麻烦, 需要一大堆代码。

示例程序 3-5 改进后的程序

```
unit Unit1;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
    Dialogs, StdCtrls, ComCtrls;  
  
type  
    TForm1 = class (TForm)  
        FontDialog1: TFontDialog;  
        btnUndo: TButton;  
        btnSet: TButton;  
        Memo1: TMemo;  
        procedure btnSetClick (Sender: TObject);  
        procedure FormCreate (Sender: TObject);  
        procedure btnUndoClick (Sender: TObject);  
    private  
        // FOriginalFont: TFont;  
        FOriginalFontSize: integer;  
        FOriginalFontName: TFontName;  
        FOriginalFontStyle: TFontStyles;
```

```

    public
    |Public declarations|
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.btnSetClick (Sender: TObject);
begin
    if FontDialog1.Execute then
        Mem1.Font := FontDialog1.Font;
    end;

procedure TForm1.FormCreate (Sender: TObject);
begin
    Mem1.Lines.Add (
        '一个可以任意设置字体和复原字体的例子。');
    //FOriginalFont := Mem1.Font;
    |保存字体的原来状态|
    FOriginalFontSize := Mem1.Font.Size;
    FOriginalFontName := Mem1.Font.Name;
    FOriginalFontStyle := Mem1.Font.Style;
    end;

procedure TForm1.btnUndoClick (Sender: TObject);
begin
    //Mem1.Font := FOriginalFont;
    |恢复字体的原来状态|
    Mem1.Font.Size := FOriginalFontSize;
    Mem1.Font.Name := FOriginalFontName;
    Mem1.Font.Style := FOriginalFontStyle;
    end;

end.

```

从示例程序 3-5 看出，这位程序员没有跳出陈旧的编程模式，没有建立真正的面向对象新思维。在示例程序 3-6 中，我们创建了一个 FOriginalFont 对象（而不仅仅是一个对象变量）来克隆原有的 Mem1 的 Font 对象，这样，Mem1 的 Font 对象所有状态和属性就保存在 FOriginalFont 对象中，需要复原 Mem1 的 Font 对象时，只要将 FOriginalFont 对象复制给 Mem1 的 Font 对象即可。奇妙的是我们在这里根本不需要知道要源对象有多少种属性以及当前的状态。

示例程序 3-6 利用克隆对象保存状态的程序

```

procedure TForm1.btnSetClick (Sender: TObject);
begin
    if FontDialog1.Execute then

```

```
    Memo1.Font := FontDialog1.Font;  
end;  
  
procedure TForm1.FormCreate (Sender: TObject);  
begin  
    Memo1.Lines.Add (  
        '一个可以任意设置字体和复原字体的例子。');  
    FOriginalFont := TFont.create;  
    FOriginalFont.Assign (Memo1.Font);  
end;  
  
procedure TForm1.btnUndoClick (Sender: TObject);  
begin  
    Memo1.Font.Assign (FOriginalFont);  
end;
```

由此可见，示例程序 3-6 以面向对象的思维方式提高了代码的效率和质量。在编程中我们经常需要对象的状态进行改变，又希望能够在不满意时将这个对象的状态复原。比如：TFont、TPen、TBrush 甚至是 TTable 这样的数据库对象。这时我们就可以利用对象克隆的技术获得该对象的一个快照 (snapshot)，以便在需要时进行复原。这一技术再结合接口、类方法、模式等面向对象技术，可能会产生更强大的功能。读者会在后面的章节看到更精彩的示例。

3.2 对象的生死

我们前面讲过，对象是通过类创建的，对象是类的动态实例。每个对象都有生命期。一个对象按其生命期来分析，一般有三个阶段，出生，活动，死亡。而我们在编程中要做的对应为：创建（初始化）、运行、销毁。所以在对象出生的时候初始化，死亡的时候销毁。

这里我们提到生命期并不是说对象有生命，而是强调对象有生死，有开始和结束。其实，万事万物又何尝不是分别以生和死作为开始和结束呢？对象也不例外，不过生成以及销毁对象都需要健全的机制做保证。否则不仅对象本身遭殃，甚至会导致程序乃至整个系统崩溃。

在 Delphi 中，对象的创建、使用和销毁都有一套完善的机制。比如：通过构造函数和析构函数来实现对象生与死的处理。理解和掌握这套机制可以减少我们管理对象生命期所带来的麻烦。

3.2.1 对象的构造和析构

编程经验告诉我们，使用一个未初始化的变量可能就是潜在的灾难，使用一个未初始化的指针将导致崩溃。所以，创建一个对象的时候就将其初始化。

```
Object.Init ();
```

同样，编程经验还告诉我们，如果一个变量不再使用就应该尽早将其销毁，以减少对内存的消耗，如果是指针引用则应该将其置空 (nil)。

```
Object.Free ();
```

虽然，我们要养成初始化变量和及时销毁变量的好习惯。但在面向对象的编程中问题却要

更加复杂,处理不好会导致致命的错误。幸好 Delphi 为我们提供了一个对象创建和销毁的机制,使我们可以轻松简单地完成对象初始化和最终的销毁工作。

在面向对象的语言中,对象生成的方式几乎都是一样的,一般流程如图 3-7 所示(VCL 类的初始化是指初始化 VMT 和接口指针)。对象一般生存在两个地方,栈或堆中。前面我们讲过,Delphi 中所有 VCL 类都保存在称为堆的自由存储区中,必须使用 Create 和 Destroy 分配、回收空间,速度自然会比存在于栈中的普通类型变量要慢一些。

Delphi 用 Constructor 声明一个构造函数,在对象产生的时候调用;用 Destructor 声明一个析构函数,将在对象销毁的时候调用。

大多数构造函数命名为 Create,但那只是一个惯例,而不是 Delphi 的要求。有时候会发现其他名称的构造函数,特别是在 Delphi 具有方法重载之前编写的旧类。为了获得与 C++ Builder 最大的兼容性,它不允许命名构造函数,应该总是用 Create 作为所有重载的构造函数,通常我们也称之为 Create 方法。

有些人可能会认为“Create 方法是每一个类都具有的隐含方法”。其实,这种说法是不准确的。事实上,在 Delphi 中所有的类都默认继承自一个最基础的类 TObject,甚至在并未指定继承的类名时也是如此。Create 方法是 TObject 类具有的方法,因此,理所当然,所有的类都自动获得了 Create 方法,不管你是否实现过它都是如此。试想,如果没有 Create 方法的话,怎样建立一个对象呢?

Create 方法是一个特殊的方法,准确地讲它是一个“Constructor”(构造函数)。通常是这样声明的:

```
Constructor Create;
```

当一个对象实例用构造函数创建时,编译器将自动对对象的每一个域进行初始化,你可以放心地认为所有数字被赋值为 0,所有指针为 nil,所有字符串为空。

当然,我们也可以通过编程来实现对象的初始化。以示例程序 2-1 的 TMan 类为例,你可能会想到增加一个 SetIni 方法来进行对象的初始化。如示例程序 3-7 所示。

示例程序 3-7 增加一个 SetIni 方法来进行对象的初始化

```
type
  TMan = class(TObject)
  private
    FAge: Integer;
    procedure SetAge(Value: Integer);
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    property Age: Integer read FAge write SetAge;
```

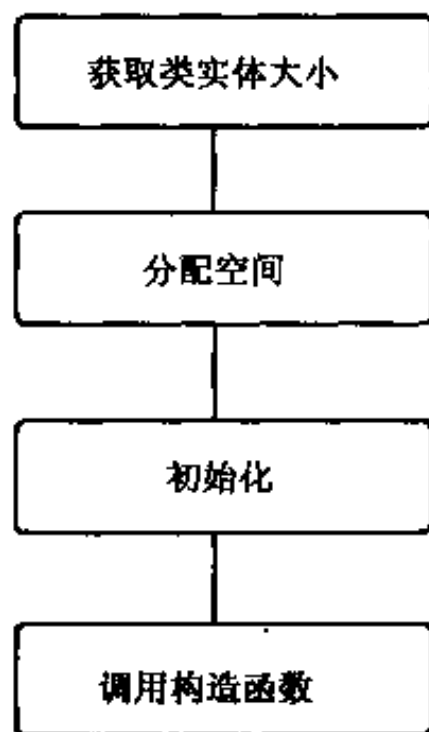


图 3-7 对象生成流程

```
    procedure SetIni (n, l, s: String; m: Boolean; a: Integer);
end;

procedure TMan.SetIni (n, l, s: String; m: Boolean; a: Integer);
begin
    Name: = n;
    Language: = l;
    SkinColor: = s;
    Married: = m;
    Age: = a;
end;
```

如下所示, 调用 TMan.SetIni 方法就可以进行初始化。

```
Var AMan: TMan;
Begin
    AMan: = TMan.create;
    AMan.SetIni ('张三','英语','黄色', False, 10);
End;
```

实际上, 就像对待一般的函数或过程那样, 我们也可以直接向构造函数传递参数, 利用 Create 方法来进行初始化。现在我将示例程序 3-7 改为示例程序 3-8 的样子。

示例程序 3-8 利用 Create 方法来进行初始化

```
type
    TMan = class (TObject)
    private
        FAge: Integer;
        procedure SetAge (Value: Integer);
    public
        Language: string;
        Married: Boolean;
        Name: string;
        SkinColor: string;
        constructor create; overload;
        property Age: Integer read FAge write SetAge;
        Constructor Create (n, l, s: String; m: Boolean; a: Integer);
    end;

procedure TMan.Create (n, l, s: String; m: Boolean; a: Integer);
begin
    Name: = n;
    Language: = l;
    SkinColor: = s;
    Married: = m;
    Age: = a;
end;
```

如下所示, 调用 Create 方法就可以进行初始化。

```
AMan: = TMan.create ('张三','英语','黄色', False, 10);
```

这样, 在 Create 方法里就完成了对数据的初始化, 而无须再调用 SetIni 方法了。

Create 方法还可以像其他方法一样实现重载和覆盖；充分利用这一面向对象的特性可以提高对象的封装和重用，增加对象初始化的技巧。

有意思的是，构造函数不仅是一个 Delphi 限定符，而且是一个 OOP 方法学的名词。与之相对应的，还有 Destructor（析构函数）。前者负责完成创建一个对象的工作，为它分配内存、初始化，后者负责销毁这个对象，回收它的内存。要注意的一点是，Constructor 的名字一般是 Create，但 Destructor 的名字却不是 Free，而是 Destroy。例如：

```
Destructor Destroy;
```

那么，为什么我们读到的代码总是使用 Free 来销毁对象呢？大家看一下基础类 TObject 的 Free 方法就明白了：

```
procedure TObject.Free;  
begin  
  if Self <> nil then  
    Destroy;  
end;
```

显然，二者的区别是：Destroy 会直接销毁对象，而 Free 会事实检查该对象是否存在，如果对象存在，或者对象不为 nil，它才会调用 Destroy。这就是说，即使 Free 调用的对象不存在，它也总是能返回成功。因此，程序中应该尽量使用 Free 方法来销毁对象，这样更加安全一些。

为确保对象即使引发异常也会正确地销毁，建议使用 try...finally 异常句柄。

例如：

```
MyObj: = TSomeClass.Create;  
try  
  MyObj.DoSomething; //可能会引起异常的方法  
  MyObj.DoSomethingElse; //其他方法  
finally  
  MyObj.Free;  
end;
```

参见 有关 try...finally 的更多信息可参见 2.5 节。

销毁一个全局对象变量时，在销毁对象时总是把该变量设为 nil，这样就不会留下一个包含非法指针的变量。特别当析构函数或者一个由析构函数调用的方法引用了该变量时，如果该变量成为了 nil，则可以避免任何潜在的问题。

注意 Free 方法也不会自动将对象置为 nil，所以在调用 Free 之后，最好是再手动将对象置为 nil。做到这点的一个简单方法是（从 SysUtils 单元）调用 FreeAndNil 过程。

```
GlobalVar: = TNewClass.Create;  
try  
  GlobalVar.Over;  
finally  
  FreeAndNil (GlobalVar);  
end;
```


通过对象的构造和析构，申请内存和释放内存的操作自动完成了，构造函数和析构函数的目的在于一个类可以像普通类型一样初始化和销毁，从而保证了封装。

注意，这里构造和析构有一个顺序问题，就是构造时应该从基类开始按继承的层次顺序调用，析构的时候顺序正好相反。这样处理是因为，派生类可能在构造函数里使用基类的成员变量，如果基类还没有创建，那就会有问题；而析构的时候，如果基类先析构，也会有这样的问题。

如果是在一个程序中，存在着容器对象包含其他对象的情况，比如：一个包含 Button 对象的 Form 对象，那么这些对象的创建和析构又是怎样完成的呢？它们之间的关系又是怎样协调的呢？

为了搞清这个问题，我们不得不简单提一下 VCL 类的组织形式：属主（owner）机制。

从 TComponent 开始，VCL 类的构造函数就带有一个 TComponent 类型的参数 AOwner。每个从 TComponent 继承的类的实体都拥有惟——一个所有者（亦即属主），这就决定了这些类之间是树形关系。例如：

```
Form1: = TForm.Create (Application);  
Form2: = TForm.Create (Application);  
Button1: = TButton.Create (TForm1);
```

这样就形成了如下以 Application 为根（Root）的树形结构，如图 3-8 所示。

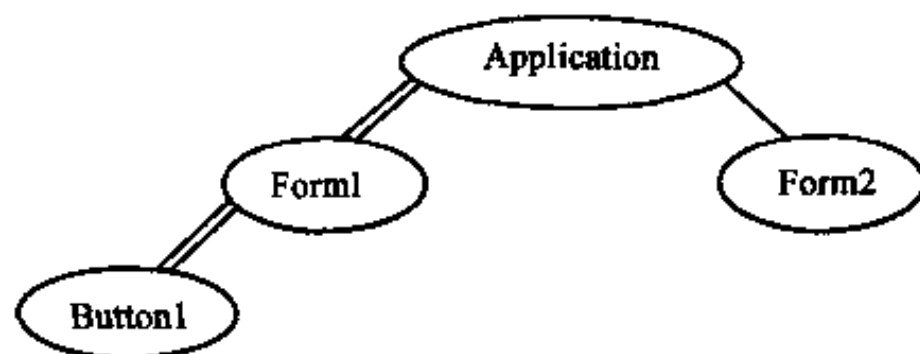


图 3-8 树形结构

图 3-8 中表示的从属关系，可以从图 3-9 中看得更明白。

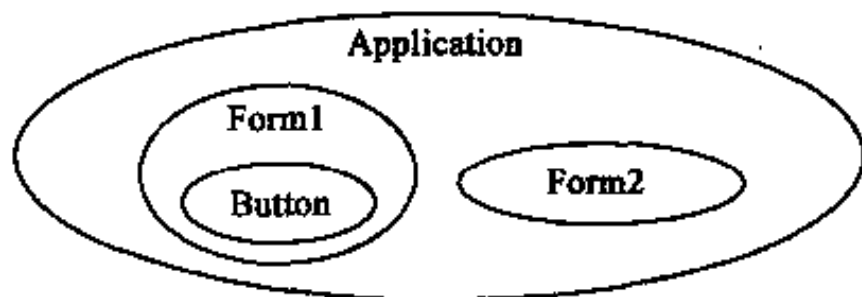


图 3-9 从属关系

于是当 Application 析构的时候，它会通知自己所拥有的对象 Form1 和 Form2 析构，Form1 和 Form2 再通知自身所拥有的对象析构，如此递归下去，如同推倒了多米诺骨牌那样。这样一来，Form1 和 Form2 以及它们所拥有的对象，不用显式调用析构函数，资源就自动回收了。

这种“属主”机制就是设计模式中典型的结构型模式 COMPOSITE（组合），即某个对象拥有一系列类似的对象，而这一系列对象中的每一个又拥有一系列的对象，如此递归下去，管理起来很方便。由 Application 向其所拥有的对象发送析构的消息的方式，是典型的行为模式

Observer (观察者), 这也是 VCL 消息处理的基本模型。

构造和析构的作用机制就是自动化, 简化编程的复杂度。还要记住的是, 在一个类的构造函数里分配了的资源尽量要记得在该类的析构函数里销毁, 当然也允许提前销毁, 你可以在析构函数里判断它是否已经销毁, 如果没有就销毁。这就是我们常说的做一件事要“善始善终”, 在面向对象编程中管理自己对象的生和死也要遵循这一原则。

参见 在 6.7 节, 我还会为大家介绍一种基于接口管理对象实例机制的垃圾对象自动回收程序。

3.2.2 如何动态生成对象

Delphi 中控制对象生成过程的代码主要在 TObject.InstanceSize、TObject.NewInstance、TObject.InitInstance 几个成员函数中。有兴趣的读者可对照下面 TObject 的声明, 源代码在 Source/Vcl/system.pas 里。分析 TObject 的源代码有助于我们了解对象的动态生成 (Dynamic Creation) 机制。

```
TObject = class
  constructor Create;
  procedure Free;
  class function InitInstance (Instance: Pointer): TObject; //初始化实例对象
  procedure CleanupInstance;
  function ClassType: TClass;
  class function ClassName: ShortString;
  class function ClassNameIs (const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: Pointer;
  class function InstanceSize: Longint; //获取实例对象空间大小
  class function InheritsFrom (AClass: TClass): Boolean;
  class function MethodAddress (const Name: ShortString): Pointer;
  class function MethodName (Address: Pointer): ShortString;
  function FieldAddress (const Name: ShortString): Pointer;
  function GetInterface (const IID: TGUID; out Obj): Boolean;
  class function GetInterfaceEntry (const IID: TGUID): PInterfaceEntry;
  class function GetInterfaceTable: PInterfaceTable;
  function SafeCallException (ExceptObject: TObject;
    ExceptAddr: Pointer): HRESULT; virtual;
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  procedure Dispatch (var Message); virtual;
  procedure DefaultHandler (var Message); virtual;
  class function NewInstance: TObject; virtual; //为新实例对象分配内存空间
  procedure FreeInstance; virtual;
  destructor Destroy; virtual;
end;
```

动态生成对象是一个相当实用的技术。比如在前面的示例程序 3-3 中, 我们提到了一个类引用的演示程序, 我们可以选择不同类型的控件, 动态地在窗体中生成。主程序对用户可能会选择哪个控件类型一无所知, 但是仍然需要“动态”地生成这些对象。又比如 Delphi 的 IDE 对

象设计器，也是一个很好的例子：双击鼠标，一个对象就在设计面板中动态生成了，可供我们设计之用了。C++ 语言本身并没有也不可能提供对动态生成的支持，不过 MFC 中用宏 (macro) 模拟就可以取得类似的效果。

仔细想来，动态生成是个很好笑的技术，它需要程序生成一个对其性质并不清楚的对象。你能造一个你不知道的东西吗？不可能。但是如果告诉你制造的原料和方法呢？那当然就很简单了。所以动态生成的关键是：留好事先约定的接口。MFC 的宏模拟就是一种方式，Delphi 则是使用了另一种方式。

高级 RTTI 方式往往会引入一个所谓“类的类”，即元类 (metaclass) 的概念。这也就是前面我们提到的类引用类型 (class-reference type)，其构造形式为：class of *type*。这里，*type* 是任何类型。

在 Delphi 中，每个类都有一个代表其相应信息的类 (见图 3-10)。代表 TObject 类信息的类是 TClass，代表 TPersistent 类信息的类就是 TPersistentClass。可以说，如果把类看做是对象的模板，那么就可以把元类看做是类的模板。

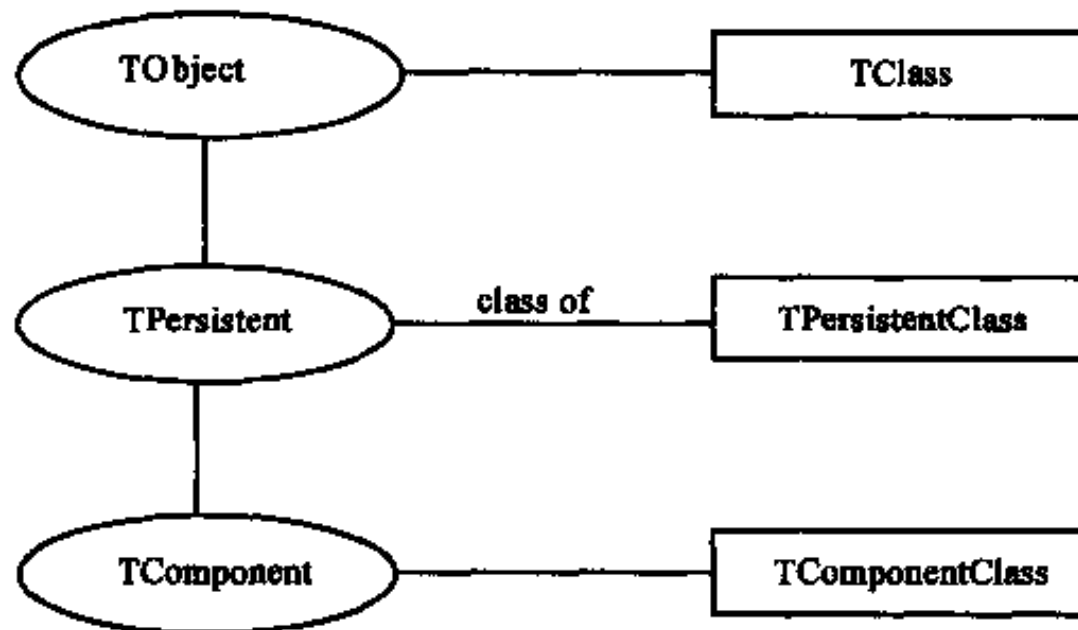


图 3-10 实现 RTTI 的元类

Delphi 可以借此来实现对 TComponent 派生类的动态生成机制：

```
function CreateComponent (AOwner: TComponent;
                          AClass: TComponentClass): TComponent;
var
    Instance: TComponent;
begin
    Instance := TComponent (AClass.NewInstance);
    {try}
        Instance.Create (Owner);
    {except}
        raise;
    end;
    CreateComponent := Instance;
end;
```

TMetaClass 究竟是什么？我们已经知道，就是指向 VMT 入口的指针。假如给我们一个 TMetaClass，我们能做到动态生成吗？对照对象生成的流程图，我们来分析以下内容：

- 获取类实体大小：通过 `TMetaClass.InstanceSize` 可以做到。
- 分配空间：这个当然可以做到。
- 初始化：如果无特殊需要，直接调用 `TObject.NewInstance` 就行了。
- 调用构造函数：TMetaClass 里可没记录过某个类的构造函数，再说，一个类的构造函数好像也不止一个吧？

惟一的问题，就出在构造函数上：我们需要一个形式固定的“虚”构造函数，也就是说留好实现约定的接口。

VCL 类从 `TComponent` 类开始，构造函数就是虚的（难怪刚才我们只生成 `TComponent` 的派生类），Delphi 所谓的虚构造是如何做到的呢？Scott Meyers 在“Virtualizing constructors and non-member functions”一文中详细说明了所谓虚构造的实现方式：事先约定一个普通的虚函数，其功能是构造函数而已。也就是说，Delphi 所谓的虚构造函数（名字是 `Create`），不过是一个事先约定好功能的普通虚成员函数罢了。在设计模式中，这叫做 `Factory Method` 模式，又名 `Virtual Constructor` 模式。

下面把示例程序 3-3 重新改写成 `Factory Method` 模式，让大家体验一下 `Factory Method` 模式是如何动态创建对象的（见示例程序 3-9）。

示例程序 3-9 `Factory Method` 模式下的动态控件创建

```
// - - - - -
//  Factory Method 模式下的动态控件创建 v1.0
//  逻辑单元 (TControlFactory)
//          刘艺 2003/02/26
// - - - - -

unit Unit2;

interface
uses
  Windows, SysUtils, Classes, Controls, Forms;

Type
  TControlClass = class of TControl;
  TControlFactory = class
  public
    class function createControlObj (AOwner: TWinControl;
                                   ControlClass: TControlClass): TControl;
  end;

implementation

class function TControlFactory.createControlObj (AOwner: TWinControl;
                                                ControlClass: TControlClass): TControl;
var
  FControlObj: TControl;
begin
  FControlObj := ControlClass.Create (AOwner);
  FControlObj.Parent := AOwner;
  FControlObj.Name := FControlObj.ClassName;
```

```

    FControlObj.SetBounds (10, 10, 250, 150 );
    result: = FControlObj;
end;

end.

// -----
//  Factory Method模式下的动态控件创建V1.0
//  界面单元( TForm1 )
//.          刘艺 2003/02/26
// -----
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, ExtCtrls, ComCtrls;

type
    TControlClass = class of TControl;
    TForm1 = class (TForm)
        RadioGroup1: TRadioGroup;
        Button1: TButton;
        Label1: TLabel;
        procedure Button1Click (Sender: TObject );
    private
    public
    end;

var
    Form1: TForm1;

implementation

uses Unit2;
{$R *.dfm}

procedure TForm1.Button1Click (Sender: TObject );
const
    CtrlClassArray: array [0..2] of TControlClass
        = (TMonthCalendar, TMemo, TColorBox );
var
    i: integer;
    ControlObj: TObject;
begin
    //清理对象
    for i: = 0 to controlcount-1 do
        if (Controls [i] is TMonthCalendar) or (Controls [i] is TMemo)
            or (Controls [i] is TColorBox) then
            Controls [i] .Free;
    //创建对象
    ControlObj: = TControlFactory.createControlObj (self,
        CtrlClassArray [RadioGroup1.ItemIndex] );

```

```

label1.Caption: = ControlObj.ClassName;
//测试对象
if (ControlObj is TMemo) then TMemo (ControlObj) .Lines.Add ('测试成功! ');
if (ControlObj is TColorBox) then TColorBox (ControlObj) .ItemIndex: = 2;
end;

end.

```

在示例程序 3-9 中，我们在新增的 Unit2 单元中声明了一个控件工厂类 TControlFactory，它是一个无数据（无状态）的类（相当于 C++ 中的静态类），用于提供一个构造控件的类方法 createControlObj。这个类方法是创建对象的接口，其目的就是创建目标类的实例对象。Factory Method 模式定义一个用于创建对象的接口，让派生类决定将哪一个类实例化。使用 Factory Method 的好处是使一个类的实例化延迟到其派生类。特别是在编译前无法确定要将哪一个子类进行实例化时，这个办法很管用。

所以分析函数 createControlObj (AOwner: TWinControl; ControlClass: TControlClass): TControl 的两个参数 AOwner: TWinControl 和 ControlClass: TControlClass，我们知道 AOwner 参数决定了控件对象的属主，即放置该对象的容器对象（例中是 Form1），而 ControlClass 参数则是控件类的元类类型（TControlClass = class of TControl;），通过 ControlClass 参数传递类引用，决定着将要实例化的控件，这些被实例化的控件都是 TControl 的派生类。如果我们需要的话，完全可以扩展该函数，以构造更多的实现方式。函数 createControlObj 也不限于仅有的两个参数，如果需要可以附加更多的参数，并重载（overload）这个函数。

Unit1 单元中的一条语句就实现了对象的动态创建：

```

ControlObj: = TControlFactory.createControlObj (self,
                                                ControlClassArray [RadioGroup1.ItemIndex] );

```

而且 TControlFactory 类无需实例化就可以调用它的类方法 createControlObj，其完全取代了示例程序 3-3 中的 TForm1.createControlObj 方法。

大家可能发现作为构造函数的 create 和类方法的使用非常相似，也是不需要实例化就可以直接使用的。其实构造函数与类方法密切相关，我们可以定义类方法来模拟构造函数的行为，并返回对象实例。

```

.....
class function TMyObj.Create: TMyObj;
begin
    result: = inherited create;
end;

procedure TMyObj.SayHello;
begin
    ShowMessage ('Hello World! ');
end;
.....

```

如果我们在程序中使用变量 MyObj: TMyObj，调用 MyObj: = TMyObj.Create 看起来就和使用构造函数一样，但实际调用的是类方法 create，该方法调用了从 TObject 继承的构造函数。类

方法 TMyObj.Create 示范了构造函数的行为。要定义一个真正的构造函数必须要用限定符 constructor 代替通常的 procedure 或 function，而且按照惯例构造函数命名为 Create。

下面我们不妨看一下示例程序 3-10。在 Unit3 单元中，我用 TControlFactory 的构造函数创建了 ControlObj 对象。而这个对象正是我们要动态创建的那个控件对象。不相信的读者可以运行该程序 (ControlFactory2.exe)，检查一下测试结果。

```
ControlObj := TControlFactory.create (self,
                                     ControlClassArray [RadioGroup1.ItemIndex]);
```

ControlObj 这个 TControlFactory 的实例对象怎么可能是动态创建的控件对象呢？难道 TControlFactory 有七十二变的本事吗？

示例程序 3-10 Factory Method 模式下动态控件创建的进一步改进

```
// - - - - -
// Factory Method 模式下的动态控件创建 v2.0
// 逻辑单元 (TControlFactory)
// 刘艺 2003/02/26
// - - - - -

unit Unit4;

interface
uses
  Windows, SysUtils, Classes, Controls, Forms;

Type
  TControlClass = class of TControl;
  TControlFactory = class
  private
    FControlObj: TControl;
  public
    constructor create (AOwner: TWinControl; ControlClass: TControlClass);
  end;

implementation

constructor TControlFactory.create (AOwner: TWinControl;
                                   ControlClass: TControlClass);
begin
  FControlObj := ControlClass.Create (AOwner);
  FControlObj.Parent := AOwner;
  FControlObj.Name := FControlObj.ClassName;
  FControlObj.SetBounds (10, 10, 250, 150);
  self := TControlFactory (FControlObj);
end;

end.

// - - - - -
// Factory Method 模式下的动态控件创建 v2.0
// 界面单元 (TForm1)
```

```

//          刘艺 2003/02/26
//-----

unit Unit3;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, ComCtrls;

type
  TControlClass = class of TControl;
  TForm1 = class (TForm)
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    Label1: TLabel;
    procedure Button1Click (Sender: TObject);
  private
  public
  end;

var
  Form1: TForm1;

implementation

uses Unit4;
{$R *.dfm}

procedure TForm1.Button1Click (Sender: TObject);
const
  CtrlClassArray: array [0..2] of
    TControlClass = (TMonthCalendar, TMemo, TColorBox);
var
  i: integer;
  ControlObj: TObject;
begin
  //清理对象
  for i := 0 to controlcount-1 do
    if (Controls [i] is TMonthCalendar) or (Controls [i] is TMemo)
      or (Controls [i] is TColorBox) then
      Controls [i] .Free;
  //创建对象
  ControlObj := TControlFactory.create (self,
    CtrlClassArray [RadioGroup1.ItemIndex]);
  label1.Caption := ControlObj.ClassName;
  //测试对象
  if (ControlObj is TMemo) then TMemo (ControlObj) .Lines.Add ('测试成功!');
  if (ControlObj is TColorBox) then TColorBox (ControlObj) .ItemIndex := 2;
end;

end.

```

比较源代码可以发现示例程序 3-10 对示例程序 3-9 进行了改进。在 Unit4 单元中, TControlFactory 使用了构造函数 create 来动态创建控件对象。更绝妙的是通过一条转型赋值语句, 将动态创建的控件对象 FControlObj 向上转型为 TControlFactory 类型, 并赋给 TControlFactory 自身, 由 TControlFactory 来传递。这就使得 TControlFactory 有了七十二变的本事。

```
self: = TControlFactory (FControlObj);
```

这样一来, 在 Unit3 单元中的 ControlObj 变量就可以在 TControlFactory 创建时获得动态创建的控件对象的引用了。

3.2.3 对象的生命期

所有的对象都是由类创建的实例, 对象有生有死, 并在自己的生命期中存活, 发挥作用。我们在编程中使用对象, 需要了解目标对象来自哪里和有多长的生命期。有两种方法能够确定目标对象来自哪里。它们由:

- 完成初始化工作的源对象创建。
- 以参数形式接收对象的传递。

如果源对象没有创建目标对象, 那么它必须接受来自别的对象的消息。换句话说, 目标对象可被当前对象创建或由外边传递进来。无论是创建的还是接收的目标对象, 它们存在的方式通常有两种:

- 存储在变量中。
- 存储在类成员中。

如果目标对象与源对象存在合成关系 (参见 3.3 节), 即目标对象作为源对象的功能组成部分, 需要被源对象使用; 通常目标对象存储在源对象的类成员中。但如果目标对象只在方法的执行期间需要, 那么它只需存储在一个在方法中说明的局部变量中。如果方法调用结束, 那么这种变量将会消失。显然后一种情况的目标对象生命期要比前一种短得多。

如果目标对象与源对象存在合作关系, 那么目标对象需要与源对象分享生命期, 所以, 这种情况下的目标对象生命期最长。

在实际开发中, 对象的生命期还受到系统和网络的资源约束。由于每个对象都将消耗一定的系统资源, 分布式系统的对象还消耗网络的带宽, 所以在使用对象时应该尽量缩短对象的生命期, 注意及时销毁不用的对象, 释放宝贵的资源。在设计时, 要降低对象间的耦合粘度; 在使用时, 要尽可能在依赖该对象的过程中创建和销毁所依赖的目标对象, 使目标对象的生命期尽可能短。

3.2.4 组件对象生命期管理的误区

在使用 Delphi 开发应用程序的过程中, 不少程序员已经习惯于通过拖放控件来完成可视化的 RAD 开发。Delphi 强大的 RAD 功能, 减少了程序员的代码编写, 提高了开发速度; 但过分依赖这种开发方式会弱化程序员对编程的独立思考, 可能会导致难以预料的副作用。

比如, 在使用 Delphi 的组件时, 程序员习惯于将组件面板上的组件拖放到 Form 上, 进行

可视化设置。他们根本不用关心这些组件对象的生命期，因为这些组件的创建和销毁是由程序自动完成的。

可是当我们需要手工在编程中加入这些组件对象时，就不得不考虑对它们的生命期进行管理。下面是我们经常看到的一段程序，程序员通过创建一个临时的 TClientDataSet 对象 TempTable，来把一些记录保存到文件中，这些记录通常是一些状态量（见示例程序 3-11）。

示例程序 3-11 有问题的代码

```
TempTable := TClientDataSet.Create (self);
try
  with TempTable do
  begin
    LoadFromFile ('StateData');
    open;

    ...
    Edit;
    FieldByName ('Done').AsBoolean := True;
    Post;
    SaveToFile ('StateData');
  end;
finally
  TempTable.Free;
end;
```

由于 TempTable 是一个 TClientDataSet 组件对象，这里是手工加进程序的，因此这个程序员非常谨慎地使用了这样一个常用的程序套来管理对象的生命期：

```
variable := typename.Create;
try
  ...
finally
  variable.Free;
end;
```

显然，他的想法是好的，他没有忘记创建并销毁对象。但问题是 Delphi 组件对象 (TComponent) 不是普通的对象 (TObject)，它有着一些自己特殊的情况。所以，严格意义上讲，示例程序 3-11 这段代码是有问题的，并不是正确的写法。

首先，创建有属主的组件不需要手工销毁。

示例程序 3-11 中使用了 TClientDataSet.Create (self)。对于 Delphi 组件来说，它们的构造方法都继承并覆盖自祖先类 TComponent 的构造方法：

```
constructor Create (AOwner: TComponent); virtual;
```

该方法的 AOwner 参数是 TComponent 类型，它实际上是目标组件对象的属主组件对象。比如示例程序 3-11 中的 AOwner 参数是 self，它是指当前的 Form 对象。所以，每当创建一个组件对象时，通过构造方法的 AOwner 参数，Delphi 自动将目标组件对象添加到属主组件对象的内部组件对象列表上。那么这个所谓的内部组件对象列表会对目标组件对象以及它们的持有者属

主组件对象有什么影响呢？主要影响有以下三点：

- 当新的组件创建并添加到组件对象列表上时，通过调用虚方法 Notification，列表上的所有组件将获得通知。同时 Notification 将传递新组件对象的引用，并指定 opInsert 选项。Notification 方法的重要作用是帮助建立组件链接。
- 当新的组件销毁并从组件对象列表上移除时，通过调用虚方法 Notification，列表上的所有组件将获得通知。同时 Notification 将传递新组件对象的引用，并指定 opRemove 选项。
- 当属主组件被销毁时，所以列在组件对象列表上的组件将一同销毁。这就是为什么在 Delphi 中，我们无需显式销毁有属主的组件对象。

注意 关于 Notification 方法以及组件的从属关系参见 9.4 节。

分析 TComponent 的析构方法 Destroy，我们发现，如果指定了属主对象，即 FOwner < > nil，此时组件的析构方法 Destroy 会调用 RemoveComponent 方法，一一销毁组件对象列表中的所有从属组件。

```
destructor TComponent.Destroy;
begin
  Destroying;
  if FFreeNotifies <> nil then
  begin
    while Assigned (FFreeNotifies) and (FFreeNotifies.Count > 0) do
      TComponent (FFreeNotifies [FFreeNotifies.Count - 1]).Notification (Self,
        opRemove);
    FreeAndNil (FFreeNotifies);
  end;
  DestroyComponents;
  if FOwner <> nil then FOwner.RemoveComponent (Self);
  inherited Destroy;
end;

procedure TComponent.RemoveComponent (AComponent: TComponent);
begin
  ValidateRename (AComponent, AComponent.FName, '');
  Notification (AComponent, opRemove);
  AComponent.SetReference (False);
  Remove (AComponent);
end;
```

由此可见，在示例程序 3-11 中，既然已经为组件指定了属主对象 (self)，就没有必要再使用 free 来显式销毁该组件，不然该组件就可能被销毁两次。

其次，创建组件时指定属主在效率上不划算。

我们不妨来研究一下 TComponent 的有关代码，了解组件对象创建的机制。

当组件对象创建时，如果指定了属主对象，即 AOwner < > nil，此时组件的构造方法 Create 会调用 InsertComponent 方法，并将新创建的组件加入到组件对象列表中：

```
constructor TComponent.Create (AOwner: TComponent);
begin
```

```

    FComponentStyle := [csInheritable];
    if AOwner < > nil then AOwner.InsertComponent(Self);
end;

procedure TComponent.InsertComponent (AComponent: TComponent);
begin
    AComponent.ValidateContainer (Self);
    ValidateRename (AComponent, '', AComponent.FName);
    Insert (AComponent);
    AComponent.SetReference (True);
    if csDesigning in ComponentState then
        AComponent.SetDesigning (True);
    Notification (AComponent, opInsert);
end;

```

同时，通过 Notification 方法，通知列表上的所有组件。注意，Notification 方法在这里是递归调用的，所有属于属主对象的组件（包括手工创建的和可视化拖放到属主对象上的组件）都将调用该方法。另外 Notification 还是虚方法，可以在 TComponent 的派生类中不断被继承覆盖。所以，这种影响是连锁式的。假设你的 Form 拥有超过 100 个以上的组件，这种影响就会带来明显的延时，且效率低下。

如果我们不指定属主参数，而用 nil 代替，那么就可以避免调用 InsertComponent 方法，并拒绝组件对象列表，这也就免除了这种递归调用，并可明显提高运行效率。同时，不需要维护组件对象列表也节约了内存开支。

```

procedure TComponent.Notification (AComponent: TComponent;
    Operation: TOperation);
var
    I: Integer;
begin
    if (Operation = opRemove) and (AComponent < > nil) then
        RemoveFreeNotification (AComponent);
    if FComponents < > nil then
        begin
            I := FComponents.Count - 1;
            while I >= 0 do
                begin
                    TComponent (FComponents [I]).Notification (AComponent, Operation);
                    Dec (I);
                end;
            if I >= FComponents.Count then
                I := FComponents.Count - 1;
            end;
        end;
    end;
end;

```

所以，把示例程序 3-11 稍加修改如示例程序 3-12 所示，这才是比较好的正确编码。由于取消了属主对象对从属组件生命期的管理，因此我们要记住使用 Free 来销毁创建的组件对象。

示例程序 3-12 比较好的正确编码

```

TempTable := TClientDataSet.Create (nil);
try
    with TempTable do

```



```
begin
  LoadFromFile (' StateData ');
  open;

  ...
  Edit;
  FieldByName (' Done ') .AsBoolean : = True;
  Post;
  SaveToFile (' StateData ');
end;
finally
  TempTable.Free;
end;
```

另外，在创建 Form 时，也要尽量避免这样的代码：

```
with TMyForm.Create (Application) do
  try
    ShowModal;
  finally
    Free;
  end;
```

因为 Application 也是组件对象，它是 TComponent 的派生类。它的 Notification 方法不仅会调用其所属组件对象（直接影响），还会波及到该 Form 对象的所属组件对象（间接影响）。比较好的方法是写成这样：

```
with TMyForm.Create (nil) do
  try
    ShowModal;
  finally
    Free;
  end;
```

对于生命期较短的组件对象的调用，一般使用下面的程序段比较理想。这样的程序段经常用于某个方法中：

```
AComponent: = ComponentType.Create (nil);
try
  ...
finally
  AComponent.Free;
end;
```

如果创建的组件对象生命期较长，而且不是在一个地方管理的，比如在 FormCreate 方法中创建，在 FormDestroy 方法中销毁（如示例程序 7-3），则此时在中间过程使用该组件对象时可能需要判定它是否还存在，以免引用出错。

```
if Assigned (AComponent) then
begin
  .....
end;
```

在这种情况下，无属主的组件对象的销毁应该是这样：

```
AComponent.Free;  
AComponent := nil;
```

或者

```
FreeAndNil (AComponent);
```

实际上，派生于 TComponent 的对象和派生于 TObject 的普通对象还有一些在生命期管理上的差别，比如在使用接口（interface）时就存在这样的问题。

我们知道，类可以实现一个接口。凡是能实现接口的类都已经含有支持 IInterface 接口或它的派生接口的实现，并保证实现了 _AddRef、_Release 和 QueryInterface 方法。TComponent 就是这样的类：

```
TComponent = class (TPersistent, IInterface, IInterfaceComponentReference)
```

接口的生命期是由引用计数（Reference Count）控制的。当我们从一个对象中取出接口时，调用了 _AddRef 方法，使得该对象的引用计数加 1；当释放某个接口将其设为 nil 时，则调用了 _Release 方法，并使得该对象的引用计数减 1。当引用计数减至 0 时，最后调用该对象的 Destroy 方法，对象被自动销毁。

对于一个普通的实现了接口的对象而言，即使是取出接口后，未将该接口变量设为 nil 也不会造成内存泄漏。因为 Delphi 的 Method Finalization 提供了一个机制，保证在退出一个方法前将所有 IInterface 类型的局部变量设成 nil。如果该接口变量不在方法中，而是在类中，那么设成 nil 的动作将由析构方法 Destroy 代劳。这些都是 Delphi 的预设行为（属于 RTL 机制），无需用户介入。但对于 TComponent 而言，它实现了接口，却没有实现引用计数机制。这意味着从 TComponent 派生对象中可以取出接口，却不能利用引用计数自动管理接口对象的生命期。因此，如果使用 TComponent 作为基类，并实现某些接口之后，可能会因为对象生命期管理的误区，导致内存泄漏！比如以下代码：

```
Component := TComponent.Create (Nil);  
vIntf := (Component as IInterface);
```

这就是造成内存泄漏的代码。因为 Component 对象不会因为引用计数变成 0 而自动销毁，实际上这里的引用计数永远不会变成 0。编程人员必须通过手工写进 free 命令来销毁对象。但是思考一下这样的代码，情况就有所不同：

```
Component := TComponent.Create (self);  
vIntf := (Component as IInterface);
```

这段代码不会造成内存泄漏。因为 Component 对象创建时指定了属主对象。那么，它的生命期由属主对象来管理，最后由属主对象在销毁自己时负责销毁 Component 对象。

最大的问题是下面这样的代码：

```
Component := TComponent.Create (Nil);  
Component2 := TMyComponent.Create (Nil);  
Component2.SomeInterface := Component as SomeInterface;  
Component.Free;  
.....
```

```
Component2.Free;
```

前面讲过, Delphi 在析构方法 Destroy 工作时会销毁所有接口变量。由于程序中已经显式地销毁了 Component 对象, 但在销毁之前并没有将 Component2.SomeInterface 设为 nil, 这样就可能导致难以预料的问题。因为 Component2 销毁时 SomeInterface 会被设成 Nil, 其中隐含了对 Component._Release 的调用, 但是 Component 已经销毁, 结果可想而知。这个问题之所以难以预料, 是因为它不是每次都发生。是否出错取决于 RTL 内部的内存分配动作。如果当时的 SomeInterface 所指的位置碰巧有效, 就可能不会出错。如果再加上该 Component 对象刚好又是某个组件的子组件, 问题就更难发现, 因为销毁子组件的顺序可能恰好是正确的。

TComponent 未能完整实现接口的引用计数机制, 这是 TComponent 的一个弱点。在 VCL 设计之初, 设计人员可能没有考虑到当今接口技术会如此广泛应用。但这个弱点并不妨碍我们“TComponent + Interface”方式的应用。我想, 对于有一定经验的程序员而言, 只要深入地研究这个问题, 绕过组件对象生命期管理的误区, 还是可以达到目的的。WebSnap 和 BizSnap 中就通过运用一些技巧, 使得“TComponent + Interface”方式可以正常工作。

参见 关于接口的有关内容参见第 6 章。

以上知识和技巧对于我们正确使用组件对象, 管理好它们的生命期非常重要。在 Delphi 面向对象编程中, 我们可能会经常使用到 TComponent 的派生类对象, 优良的编码不仅带来了高质量的产品, 更凸显了程序员独立思考的态度和扎实过人的功底。

3.3 对象的关系

使用面向对象思维方式的程序员眼中应该只有对象, 因为在他们看来一切都是对象。对象的使用带来了编程上的效率, 他们既可以重复使用已有的现成对象, 也可以通过继承产生新的对象, 而不必从头开始写起。而面向过程程序员的思维方式则停留在函数和过程上, 通过复制和粘贴代码来节省工作量。

对象复用是面向对象编程的重要思想, 它不仅仅是为了提高编程的效率, 少写几段代码, 更重要的是对象复用体现了编写可维护、可进化、可重用的程序的设计理念。所谓可维护, 是指类本身可以单独开发、测试和维护。因为对象之间的耦合比较弱, 所以使用已经调试好的对象以及事后维护该对象都比较安全, 不需大动干戈出现牵一发而动全身的问题。由于继承的特性, 使得类是可进化的。即使在设计时无法预料新的需求变化, 也可以通过预留接口(比如: 虚方法、抽象方法)和多态来完善。对象的复用使得面向对象编程类似一种对象的装配过程, 通过对象之间的不同关系, 使对象组合、协作从而实现程序的功能。当然要完美地实现这一切还依赖于面向对象的分析和设计, 比如: 如何识别和抽象对象、如何规定对象的粒度和边界、如何建立对象的交互机制等等。在这里我们重点讨论的是 OOP, 即从编程的角度来看如何实现对象的复用, 当然也涉及到面向对象的分析和设计, 因为掌握对象复用的关键在于深刻理解对象之间的关系。

一个对象是一个类的实例, 每个类(因此也是作为类实例的对象)都具有属性(状态)和方法(动作)。一个类还可以具有属于整个类的类方法。Delphi 还可以定义接口, 以提供一组

随时可访问的变量，并确保类将执行一组指定的方法。因此，对象之间的关系是通过类之间的关系体现出来的，如图 3-11 所示。这些关系主要集中在：

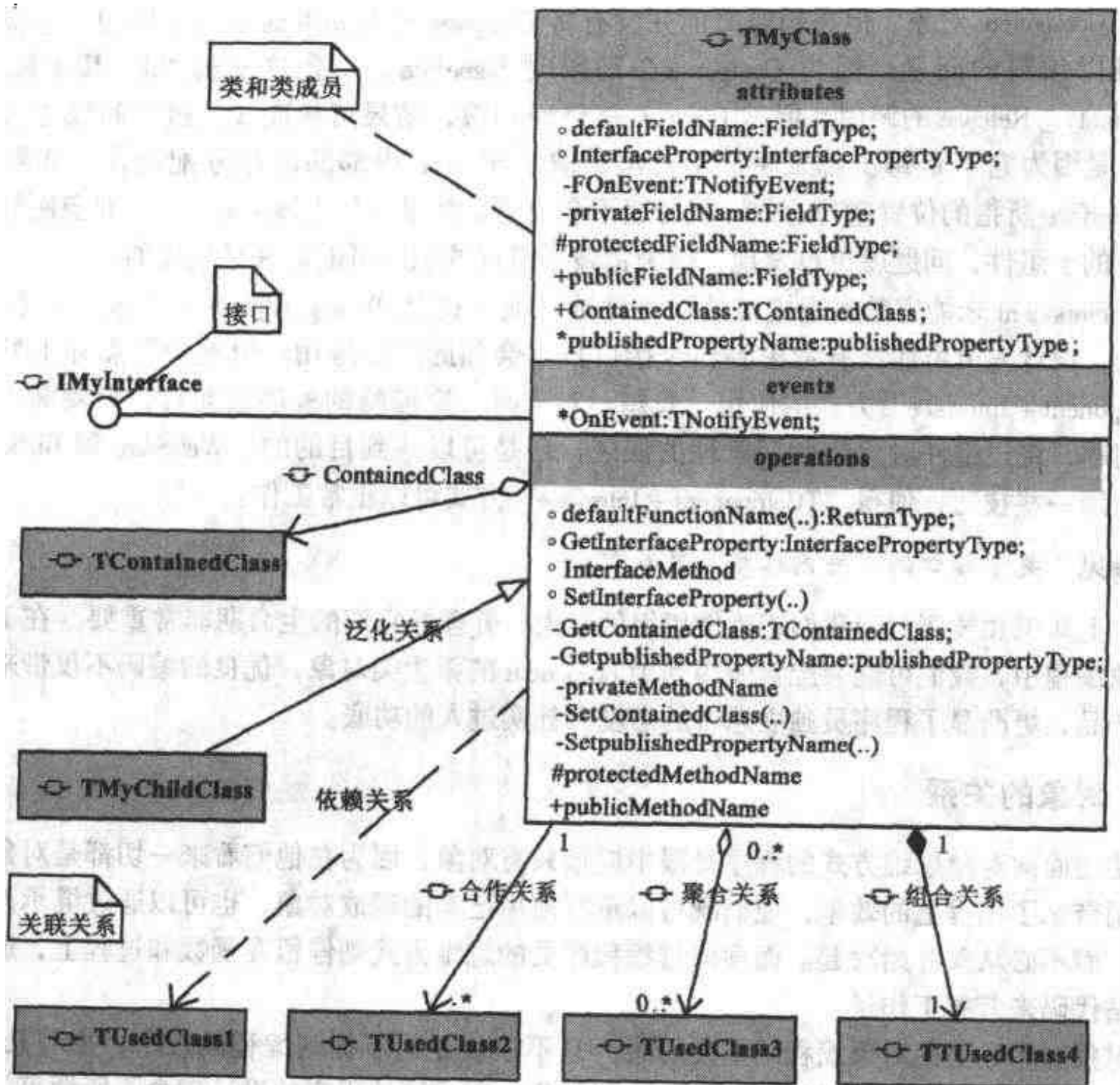


图 3-11 类（对象）关系的 UML 图示（用 ModelMaker 绘制）

- 泛化（generalization）关系：表现类（对象）的父子继承特性。
- 聚合（aggregation）和组合（composition）关系：表现类（对象）的部分和整体的合成特性。
- 依赖（dependency）和合作（association）关系：表现类（对象）的互作用特性。

这些元素之间的互相关联和作用构成了应用程序，并且也正是这些互相关联和作用使得理解面向对象编程更加困难。

注意 原始数据（简单的数据类型）和对象（类类型）间是有区别的。但这主要是对于实现上的区别。从完全面向对象程序概念化的观点来讲，原始数据可以看做是一个只有单一属性并且不具有方法的简单对象。

本节将探究对象之间的关系，包括：

- 类和对象的关系。
- 对象之间的关系基础。
- 继承关系和合成关系。
- 依赖和合作关系。

3.3.1 对象、类和类型

1. 对象

对象封装了属性和操作。在 Delphi 中，对象的属性以称为 field 的数据成员的形式存在，它决定了对象的可能状态。数据成员是使用声明性或面向数据的抽象从一个领域中提取出来的。可以是简单或复杂的数据类型。简单的数据类型不能简化为任何下级部分。复杂的数据类型是可以简化为下级部分的聚合体。数据成员可以是单值或多值的。对象的操作以称为 function 或 procedure 的对象方法的形式存在，它决定了对象可能的行为。方法是使用过程性或面向过程的抽象从一个领域中提取出来的。方法包含有名称、输入参数、输出参数，还可能有返回参数。并在接收到一条消息后被调用。

对象可以是主动的也可以是被动的。主动的对象拥有控制线程，或者可以发起活动。它们可以主动地向其他对象要求服务。被动的对象没有控制线程，不能主动地向其他对象要求服务，除非它们接到了来自主动对象的控制。

对象有生命期，可以是长久的也可以是暂时的。长久的对象在它们的创建者不复存在时仍然存在。暂时的对象仅在它们的创建者存在的时候存在。对象可以被引用，即表示为一些其他的方式。

对象还具有身份标识。所有的对象都是独一无二的，可以与其他对象区分开来。可以对对象进行下列比较：

- 完全一致的身份标识，当对象是同一对象时为真。
- 浅度相等，当对象具有相同的属性值时为真。
- 深度相等，当对象的各个下级部分均具有相同的属性值时为真。

对象之间存在相互作用的关系，对象参与事件。对象具有语义，即它有一定的意义或用途。对象是类的实例。对象及其类之间的关系视为是一种“is-a”（类属）的关系。

2. 类

类是对具有共同实现的一些对象或一系列对象的描述。类关注共同结构特征或行为特征的实现。类具有内部含义或能力。这是类定义其对象的模式的能力。类决定这些对象的结构和行为。类同样具有外部含义或能力。这是类创建类对象的能力。类被称为对象工厂。外部能力也指类引用该类的所有对象的能力。类引用（类的类）称为类的扩展。

Delphi 的类可以封装称之为类方法的类的行为。类方法由类作为方法或子程序实现。被认为是由类（或类的对象）提供的服务。类方法既可以通过对象实例调用，也可以通过类本身引用。Delphi 的类还可以有抽象方法来封装类的抽象操作。即说明一个接口而不定义实现。当类

具有一个或多个抽象方法时，它就是不能有任何实例的抽象类。一旦一个子类为所有的抽象方法提供了实现，则该子类可以有实例。当一个类的所有方法都已定义，则它属于具体类。如果类的一个方法是抽象方法，那么类可以使用该方法，但是方法的实现要延迟到子类为之提供覆盖方法。

类定义了类成员在类的对象外部的可访问性。Delphi 有四种类成员的可访问性，分别为 `published`、`public`、`protected`、`private`。

类具有与其对象共享的生命周期，如图 3-12 所示。类之间同样具有关系。类也参与事件，类实际上为其对象定义了一个实现。

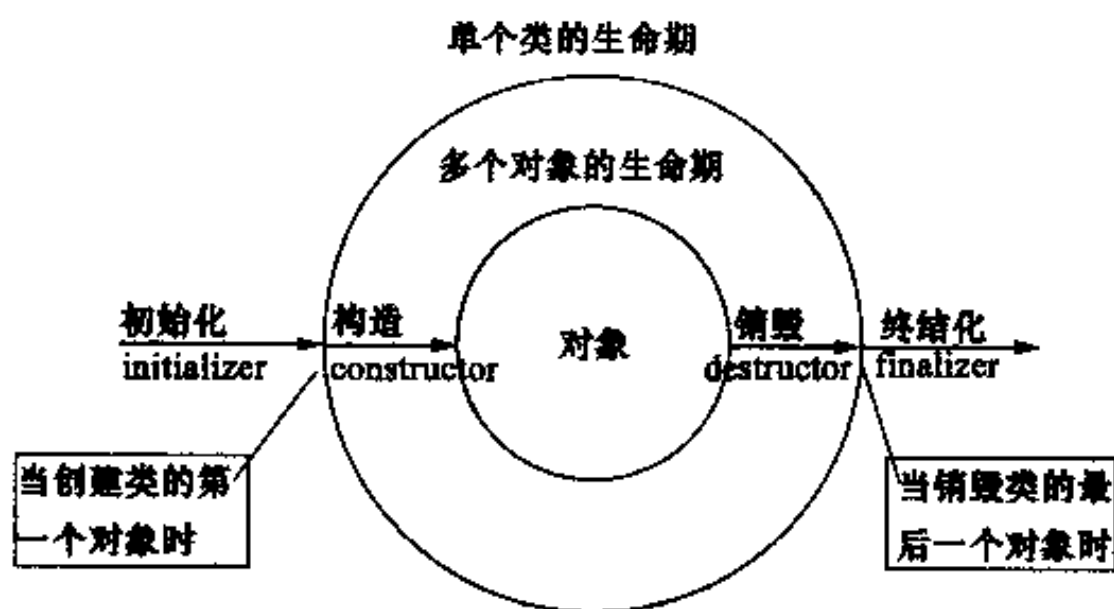


图 3-12 类和对象的生命期

3. 类型

类型是对具有公共说明或接口的对象或一系列对象的描述。类型关注统一的结构特征和行为特征的说明。类型在定义从对象或类外部对属性和操作进行访问的可访问性时，使用和类相同的可访问性标准。类型可以显式地与类相关。即对象的类可以与对象的类型不同。其实现方式是一个类从类型接收一个接口，并且提供该接口的实现。Delphi 有限支持编译期类型检查，在 Delphi 中关于类型的兼容与转换通常要用到转型技术（后面章节会详细讨论）。类型也可以隐式地与类相关，即一个对象的类隐式地包括它的类型（此时对象的类与对象的类型一致）。其实现方式是通过类的内部表示，以及由类定义一个接口并且提供其实现。

4. 对象和类的层次

类是生成对象的模板。它是可用来创建对象的蓝本或模型。运行时，相互作用的正是对象。类定义用来创建新对象，但在此之后，它们几乎无法接受程序的操作。不管组成对象的方法和属性是作为该对象类的一部分定义的，还是直接从一个基础类声明而来的，都没有什么关系，对象行为是一样的。

通过类的继承关系，我们可以看到对象是由哪一层次上的类创建的。换句话说，类层次是创建新类的方便结构，但只要是涉及到对象时，它几乎可被认为是无关的。

Delphi 的开发环境允许用户“分解”类定义，这样可以看到所有与类相关的操作和属性，而不管它们位于类层次的什么位置。

通过 Delphi 的类浏览器可以实现这些。如图 3-13 所示可以在 Exploring Classes 窗口看到树

状结构的类层次图，这将让我们看到定义在类层次中的所有类成员以及它们从何而来。从 TForm 类上溯到 TObject 类。Delphi 开发环境可让用户看到任意类这一级的详细信息。

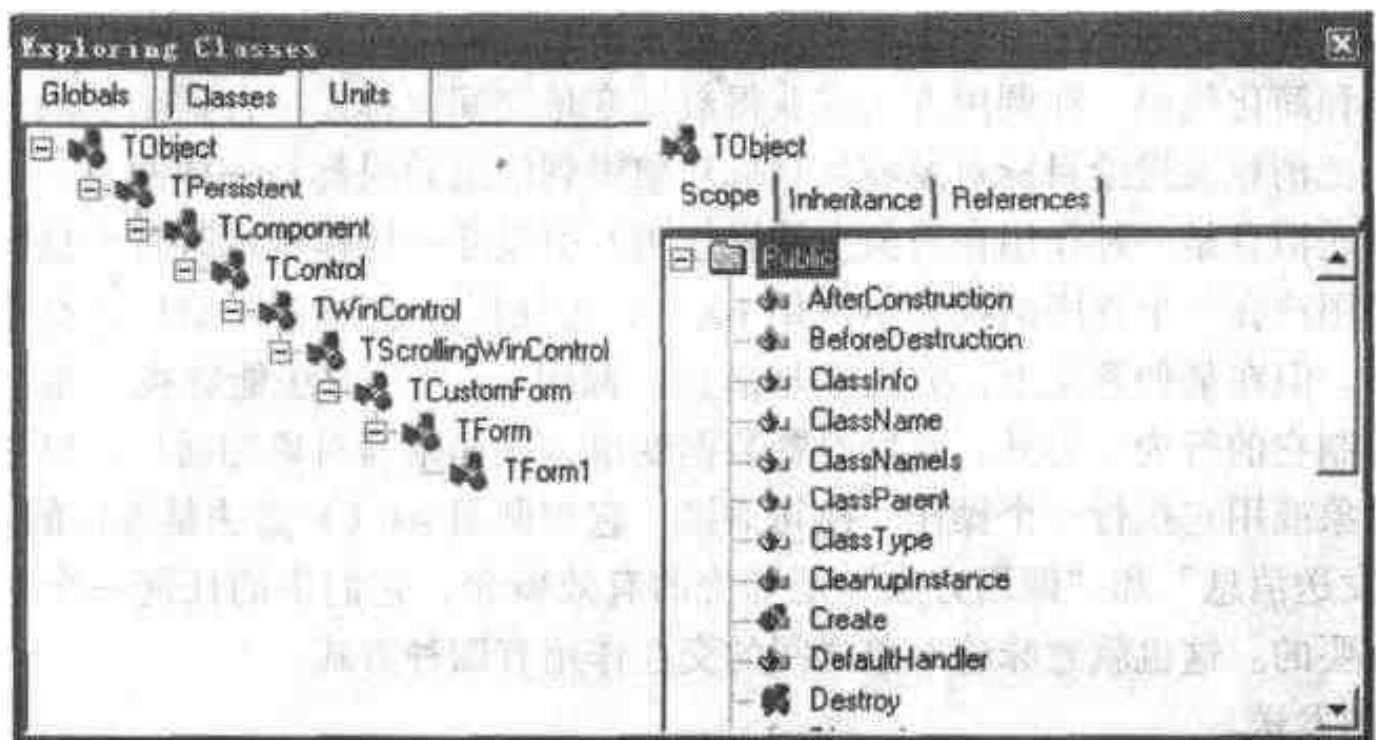


图 3-13 Exploring Classes 窗口给出了 TForm1 类的一些信息，以及它的继承关系

至此，读者已将静态类层次从运行对象层次中分离了出来。当涉及到一个应用程序时要同时考虑这两个层次。你所需的对象将提示你需要定义的类，并且所需的类之间的相似性将提示你指定通用元素所需的高级类。

3.3.2 对象之间的关系基础

对象之间是以何种方式交互作用的？对象之间的关系是建立在什么基础之上的？答案是消息。Roger S. Pressman 在《软件工程：实践者的研究方法》中称“消息是对象间交互的手段”，“消息刺激接受对象的某种行为发生，在操作执行时行为完成”。这就是说，消息传递将一个面向对象的系统中的各个对象连接在一起，通过消息我们可以观察个体对象的行为以及整个面向对象系统。

实际上一个对象对另一个对象可做的只有三件事。一个对象可以：

- 发送消息给另一个对象。
- 从另一个对象获取消息。
- 将对象传递给一些其他对象。

在面向对象编程中，操作被发向一个特定对象的服务请求（即消息）所激发；而面向过程的编程中，操作是通过特定数据进行函数调用的。前者的侧重点放在对象上，后者则放在函数上。消息传递是一种功能强大的机制，它为我们提供了重载名称和复用软件的能力，因为对消息的解释因不同类的对象而异。这一点是函数调用所无法做到的。

一个对象可以通过直接把数据存储在对象变量中或者通过把消息作为参数传递以调用方法，来把信息“发送给”另一个对象（目标对象）。类似地，目标对象通过直接接受对象变量或者通过调用方法来获得消息，从而也可以接受信息。

如果用户直接获得对象变量或调用方法与目标对象来交换数据，效果是相同的（即都能得

到信息)。这样会得到很好的灵活性,即用很基本的方法就能使每个对象的状态得到改变,而不会影响依赖它的任何对象。

另一方面,对于在同一个单元中相互工作的紧密结合的类而言,用户可以直接得到数据变量来提高性能和简化代码。在调用方法或获得数据变量之间选择是一种辅助手段。从程序分析的观点来看,把消息发送给目标对象或从目标对象得到消息效果都是一样的。

向对象发送消息是一种作用在对象上的操作吗?引发某一操作构成某种消息吗?

例如,当用户在一个程序的线程中调用 `run()` 方法时,通知它开始独立操作,没有消息传递给其形参。但在某种意义上,方法就是消息。调用 `run()` 方法能够视为将消息传递给程序,由此来控制它的行为。另外,直接设置数据变量,比如读写对象的属性,可被视做为发送消息给目标对象或用它执行一个操作。换句话说,它和调用 `set()` 方法是等价的。

另外,“发送消息”和“调用方法”是等价的有效概念,它们中的任何一个对于编程的理解都是非常重要的。这也就意味着对象之间的交互作用有四种方式:

- 数据的发送。
- 数据的接收。
- 触发操作。
- 消息传递给其他对象。

3.3.3 对象的继承与合成

对象的继承是一种在保持对象差异的同时共享对象相似性的复用。它是源自类的泛化机制。这种抽象机制允许类之间共享代码,这大大减少了代码长度并且使软件易于维护。对象通过继承,保证了实现部分紧内聚和松耦合的良好特性。通过继承我们可以拥有这样的能力:一旦定义了车辆的特征,之后通过增加不同新方法和新属性就能定义自行车和汽车。显然,新对象都继承了车辆的轮子属性,从而实现了有关车轮的复用。

在 UML 中,我们用一个从子指向父的箭头来表示泛化关系,箭头为一个空心三角形。多个泛化可以用箭头组成的树来表示,每一个分支指向一个子类,如图 3-14 所示。

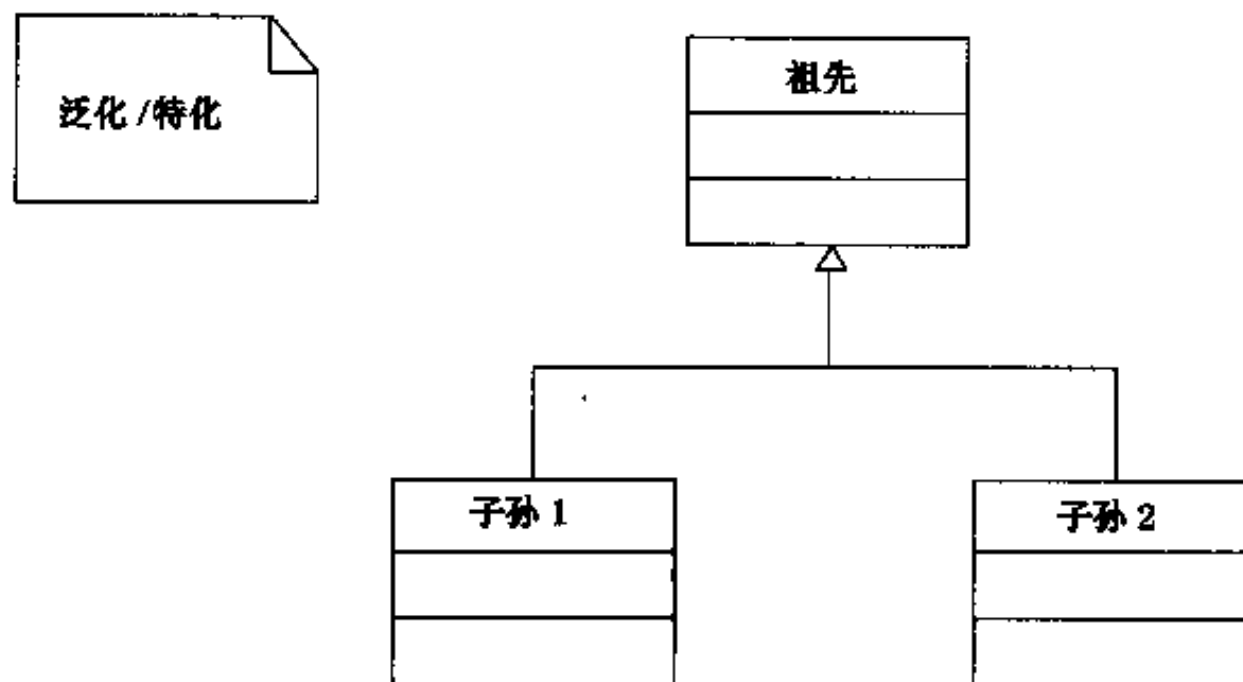


图 3-14 泛化/特化关系的 UML 表示

对象的合成是一种直观复用对象的方法，即新的对象是由已有的对象组合而成。正如 Bruce Eckel 认为的那样，这种情况只是“单纯地重复运用既有的程序代码功能，而非重复运用其形式”。

合成不同于继承，它表示了整体与部分的关系。比如：文档对象是由图元对象、文字对象组合而成。房子的组成材料包括了石材和木材。但是你不能说图元是从文档继承而来，或石材是房子的派生类。

对象的合成关系又分成聚合（aggregation）关系和组合（composition）关系两种。

聚合是描述整体/部分关系的关联与链接。聚合关系体现为“has - a”（具有）关系。是实体之间的具体或概念上的整体/部分关系的抽象。它既包括与其组成部分相连接的聚合体或整体，也包括独立于它们的聚合体存在的组成部分。聚合关系是可以传递的，即如果 A 是 B 的一部分，且 B 是 C 的一部分，则 A 是 C 的一部分。聚合关系还是反对称的，即如果 A 是 B 的一部分，那么 B 不是 A 的一部分。

组合是具有强所有权和一致生存时间约束的聚合，也称为组合聚合（composite aggregation）。组合关系体现为“contains - a”（包含）关系。它是对实体间的具体或概念上的整体/部分关系的抽象，并且这种关系在实体间具有所有权和一致生存时间约束。组合关系指定组合体或聚合体拥有它们的组成部分，组成部分只能有一个属主（owner），组成部分与其组合体属主一起存在、生存或灭亡。

在 UML 中，我们使用端点带有实心菱形的线段来表示组合关系，表示部分对象只能属于一个整体，并且通常认为部分与整体同生同灭。而对于关系比较松散的聚合关系，使用端点带有空心菱形的线段来表示，如图 3-15 所示。有时我们会在线段的一端加上箭头表示航向。

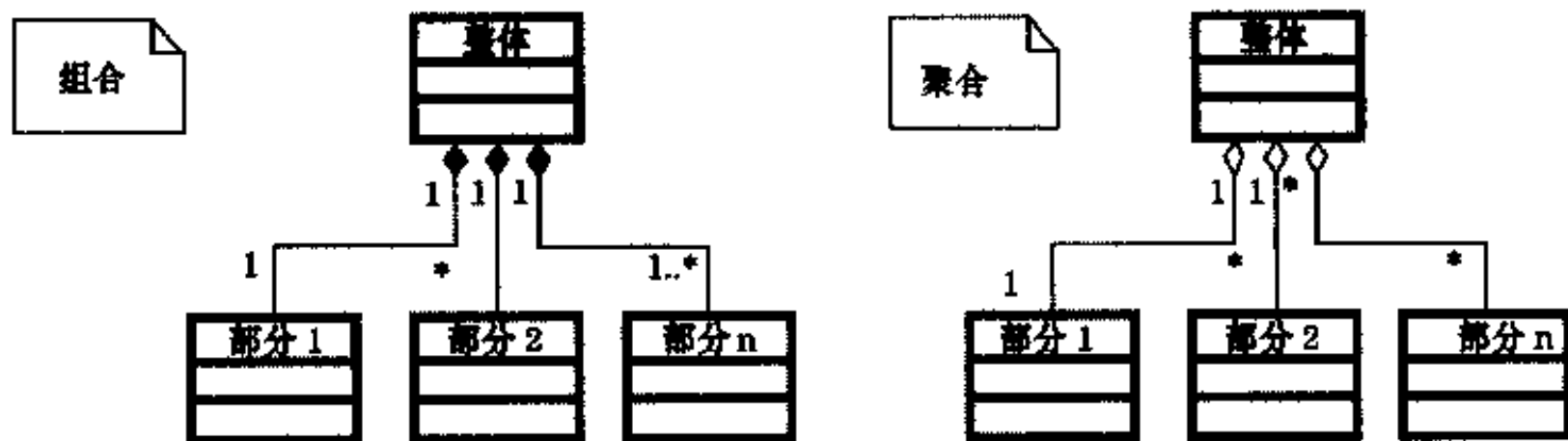


图 3-15 组合和聚合关系的 UML 表示

对象的继承关系和合成关系是面向对象系统中的重要基本关系，实际开发中这些关系往往是错综复杂的，需要程序员有足够的认识和把握能力来处理这些关系。下面我专门设计了一个示例程序来帮助大家认识和使用对象的继承关系和合成关系。

图 3-16 显示了一个车辆系统的继承关系和合成关系。这里自行车和汽车是车的派生类，体现了继承关系；而车的零部件（如：车轮）则与车构成合成关系。有些通用的零部件（如：车轮）是可以随着基类车继承下来，被自行车和汽车所用；有些专用的零部件（如：发动机）不能继承自基类，只能为某一派生类（如：汽车）所用。这样一来，我们可以看到汽车的零部件包括发动机和车轮，其中车轮是从车继承下来的。另外，即使是随基类继承下来的组合对象，还存在在派生类中可使用数目上的差异，比如：汽车使用 4 个轮子对象，而自行车是 2 个。

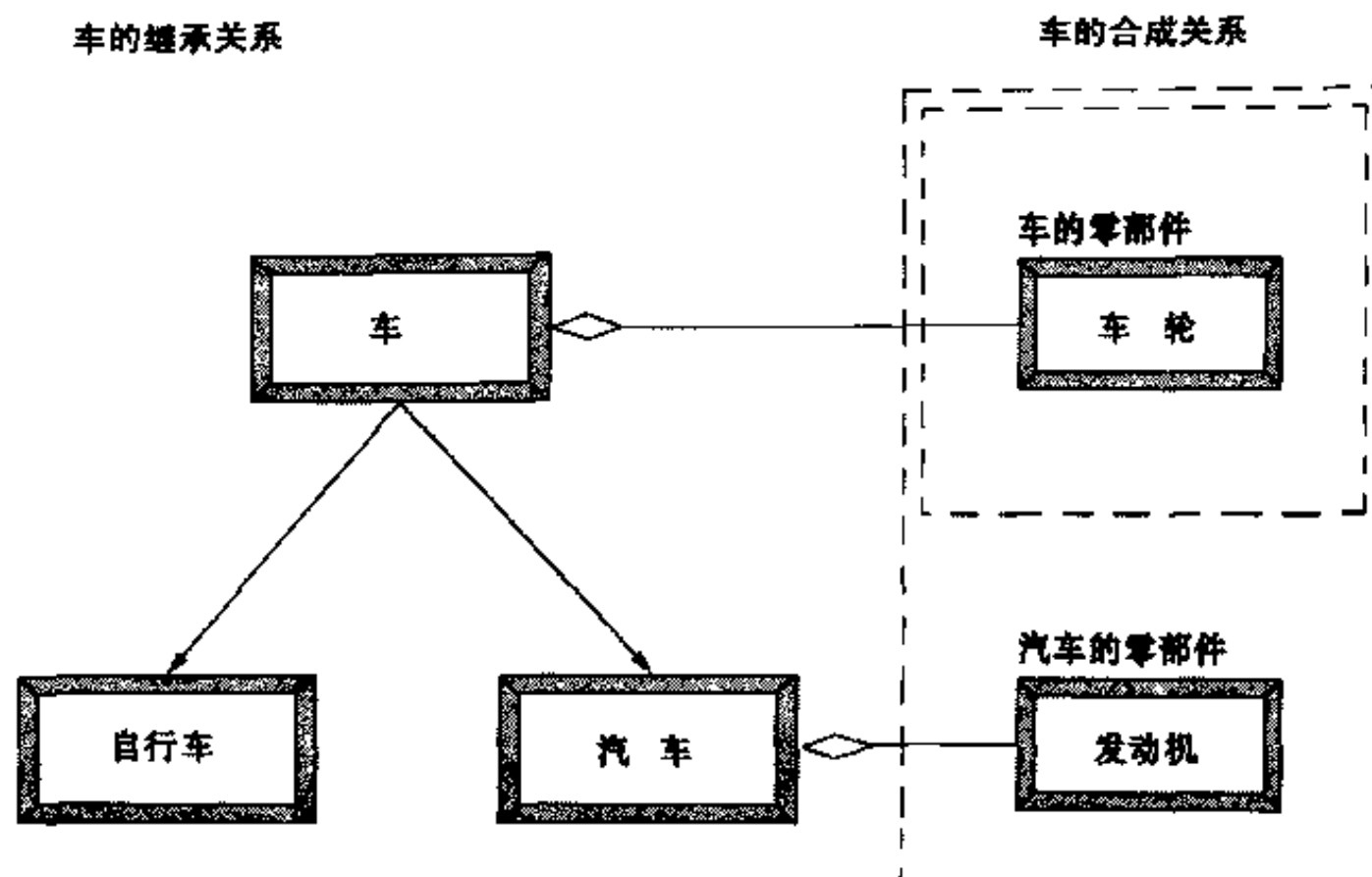


图 3-16 车的继承关系和合成关系示意图

搞清楚车的继承关系和合成关系之后我们来看看如何用 Delphi 来实现这个程序。图 3-17 是这个示例程序的 UML 类图。从图上我们可以清楚地了解系统中使用的类以及类之间的关系。

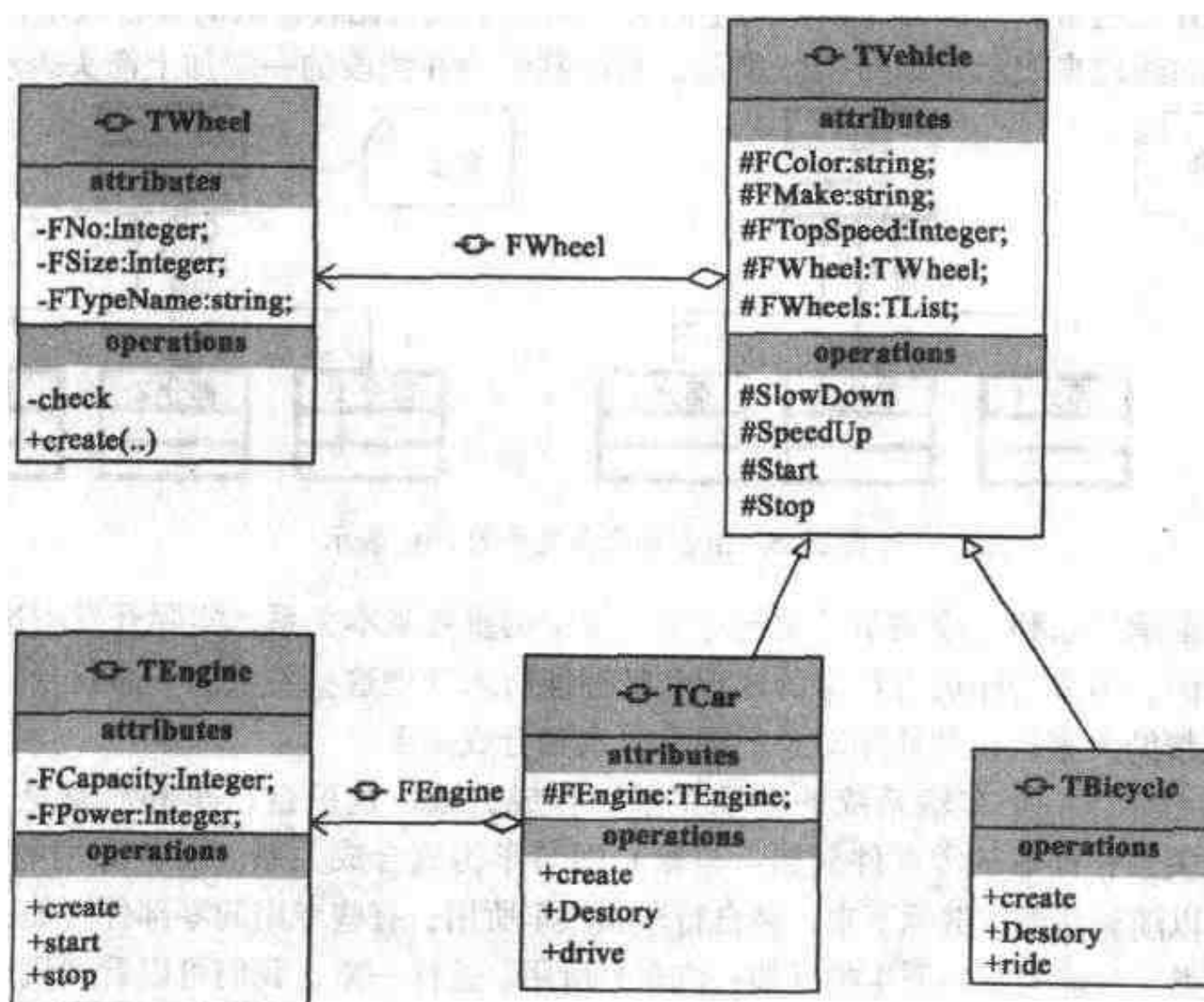


图 3-17 车的继承关系和合成关系示例程序 UML 图

首先, TVehicle 中包含了车的一些基本属性和方法, 为了继承这些属性和方法, 类成员都放在了 Protected (#) 和 Public (+) 域中。TCar 和 TBicycle 是 TVehicle 的两个派生类, 除了继承 TVehicle 的启动 (Start)、停止 (Stop)、加速 (SpeedUp)、减速 (SlowDown) 外, 它们还有着自己的行驶方法: 骑自行车 (ride) 和驾驶汽车 (drive)。

TVehicle 和 TWheel 之间、TCar 和 TEngine 之间存在着合成关系。不同的是 TCar 和 TBicycle 还继承了 TVehicle 和 TWheel 之间的合成关系, 这种关系是通过 TVehicle 类中 TWheel 类型的数据成员 FWheel 体现出来的, FWheel 通过继承可以被 TVehicle 和 TWheel 使用。

通过下面的示例程序 3-13 我们可以看到使用 Delphi 是如何编程实现车的继承关系和合成关系的。读者可以在随书光盘中找到这个程序的项目文件和所有源码, 并可通过运行程序体会各个对象运行时的交互关系。

示例程序 3-13 车的继承关系和合成关系示例程序

```
// - - - - -
// 汽车的继承关系和合成关系示例程序 V3.0
// 业务逻辑单元 (Demo.pas): 该单元包含了汽车及其零部件对象
// 刘艺 2003/04/04
// - - - - -

unit Demo;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TEngine = class (TObject)
  private
    FCapacity: Integer;
    FPower: Integer;
  public
    constructor create;
    procedure start;
    procedure stop;
  end;

  TWheel = class (TObject)
  private
    FNo: Integer;
    FSize: Integer;
    FTypeName: string;
    procedure check;
  public
    constructor create (size: Integer; TypeName: string; No: Integer);
  end;

  TVehicle = class (TObject)
  protected
```

```

    FColor: string;
    FMake: string;
    FTopSpeed: Integer;
    FWheel: TWheel;
    FWheels: TList;
    procedure SlowDown;
    procedure SpeedUp;
    procedure Start;
    procedure Stop;
end;

TBicycle = class (TVehicle)
public
    constructor create;
    destructor Destroy;
    procedure ride;
end;

TCar = class (TVehicle)
protected
    FEngine: TEngine;
public
    constructor create;
    destructor Destroy;
    procedure drive;
end;

implementation

{
    * * * * * TBicycle
    |
    constructor TBicycle.create;
    var
        i: Integer;
    begin
        FColor: = '白色';
        FMake: = '永久';
        FTopSpeed: = 20;
        ShowMessage (FColor + FMake + '车,最高时速: ' + IntToStr (FTopSpeed) );
        FWheels: = TList.Create;
        for i: = 1 to 2 do
            FWheels.Add (TWheel.create (21, 'B型自行车车轮', i) );
        end;

        destructor TBicycle.Destroy;
        var
            i: Integer;
        begin
            for i: = 1 to 2 do
                TWheel (FWheels.Items [i]) .Free;
            inherited;
        end;
    }

```



```

procedure TBicycle.ride;
begin
  start;
  speedUp;
  showmessage ('自行车行驶中 ...');
  SlowDown;
  Stop;
end;

{
  * * * * * TCar
}
constructor TCar.create;
var
  i: Integer;
begin
  FEngine := TEngine.create;
  FColor := '黑色';
  FMake := '红旗';
  FTopSpeed := 200;
  ShowMessage (FColor + FMake + '车,最高时速: ' + IntToStr (FTopSpeed));
  FWheels := TList.Create;
  for i := 1 to 4 do
    FWheels.Add (TWheel.create (36, 'A型汽车车轮', i));
end;

destructor TCar.Destroy;
var
  i: Integer;
begin
  for i := 1 to 4 do
    TWheel (FWheels.Items [i]).Free;
  FEngine.Free;
  inherited;
end;

procedure TCar.drive;
begin
  start;
  FEngine.start;
  speedUp;
  showmessage ('汽车行驶中 ...');
  SlowDown;
  FEngine.stop;
  Stop;
end;

{
  * * * * * TEngine
}
constructor TEngine.create;
begin
  Fcapacity := 1500;
  Fpower := 130;

```

```

end;

procedure TEngine.start;
begin
    ShowMessage (inttostr (Fcapacity) + 'cc,' + inttostr (Fpower)
                  + '匹马力的发动机发动了');
end;

procedure TEngine.stop;
begin
    ShowMessage ('发动机关闭了');
end;

|
***** TVehicle
|
constructor TVehicle.create;
begin
    FColor: = '';
    FMake: = '';
    FTopSpeed: : = 0;
    FWheel: = nil;
    FWheels: = nil;
end;

procedure TVehicle.SlowDown;
begin
    ShowMessage ('正在减速...');
end;

procedure TVehicle.SpeedUp;
begin
    ShowMessage ('正在加速...');
end;

procedure TVehicle.Start;
begin
    ShowMessage ('车子开始启动');
end;

procedure TVehicle.Stop;
begin
    ShowMessage ('车子停下');
end;

|
***** TWheel
|
constructor TWheel.create (size: Integer; TypeName: string; No: Integer);
begin
    FSize: = size;
    FTypeName: = TypeName;
    FNo: = No;
    check;

```

```
end;

procedure TWheel.check;
begin
    ShowMessage ('检查第' + IntToStr (FNo) + '个车轮。型号: ' + FTypeName
                + ' 大小: ' + IntToStr (FSize) );
end;

end.

// - - - - -
// 汽车的继承关系和合成关系示例程序 v3.0
// 用户界面单元( frmDemo.pas)
//                刘艺 2003/04/04
// - - - - -
unit frmDemo;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, jpeg, ExtCtrls;

type
    TForm1 = class (TForm)
        Button1: TButton;
        Button2: TButton;
        Image1: TImage;
        Button3: TButton;
        procedure Button2Click (Sender: TObject);
        procedure Button1Click (Sender: TObject);
        procedure Button3Click (Sender: TObject);
    private
        {Private declarations}
    public
        {Public declarations}
    end;

var
    Form1: TForm1;

implementation

uses Demo;

{$R *.dfm}

procedure TForm1.Button2Click (Sender: TObject);
var MyCar: TCar;
begin
    MyCar := TCar.Create;
    try
        MyCar.drive;
    finally

```

```
        MyCar.Free;  
    end;  
end;  
  
procedure TForm1.Button1Click (Sender: TObject);  
var Bicycle: TBicycle;  
begin  
    Bicycle := TBicycle.Create;  
    try  
        Bicycle.ride;  
    finally  
        Bicycle.Free;  
    end;  
end;  
  
procedure TForm1.Button3Click (Sender: TObject);  
begin  
    close;  
end;  
  
end.
```

这里需要读者注意的是，在对象的合成关系中，被使用的部分对象，如这里的 `TWheel` 和 `TEngine` 对象，需要与整体对象发生关系，因此它们的生命期和整体对象的生命期需要重叠。那么它们由谁来创建和销毁？何时创建和销毁？这是什么关系？这就是我们下面要讨论的问题。

我们知道对象的合成关系有两种，一种是组合，一种是聚合。前者意味着部分是整体不可缺少的部分，这些部分对象组成了整体对象，并只能为该整体对象独占。这就是说，在组合关系中要求由代表整体的对象负责代表部分的对象的生命周期，部分对象的生命期依赖于整体对象的生命期，整体对象负责部分对象的生死。比如：车轮和发动机是组成汽车不可缺少的部分，汽车对象实例在创建过程中同时负责创建车轮和发动机对象实例。但不同的汽车对象实例（如：红旗车、夏利车就是不同的汽车对象实例）只拥有自己的车轮和发动机对象实例。以发动机为例，显然不同的对象实例不能共享一个发动机对象实例（实际生活中我们也没有见过多辆车共用一个发动机）。当汽车对象实例销毁时，也就同时销毁了车轮和发动机对象实例。

在后者的聚合关系中，对象之间的关联要比组合关系弱。部分对象不是整体对象专属的，部分对象可以共享。比如车用 GPS 导航系统，是汽车可有可无的配备，而且 GPS 导航系统可以用于多个汽车对象。GPS 导航系统的生命期无需依赖于汽车对象的生命期，无论有无 GPS 导航，汽车对象都可以使用。

图 3-18 清楚地表示出了有关汽车聚合关系和组合关系的 UML 类图。在图中 GPS 导航机与汽车是聚合关系，车轮和发动机与汽车是组合关系。

那么，在 Delphi 中如何去实现聚合关系和组合关系呢？我们通过下面的示例程序 3-14 来给读者演示。


```
TEngine = class (TObject)
private
  FCapacity: Integer;
  FPower: Integer;
public
  procedure start;
  procedure stop;
end;

TWheel = class (TObject)
private
  FNo: Integer;
  FSize: Integer;
  FTypeName: string;
  procedure check;
public
  constructor create (size: Integer; TypeName: string; No: Integer);
end;

TVehicle = class (TObject)
protected
  FColor: string;
  FMake: string;
  FTopSpeed: Integer;
  FWheel: TWheel;
  FWheels: TList;
  procedure SlowDown;
  procedure SpeedUp;
  procedure Start;
  procedure Stop;
end;

TBicycle = class (TVehicle)
public
  constructor create;
  destructor Destroy;
  procedure ride;
end;

TCar = class (TVehicle)
private
  FGPSReceiver: TGPSReceiver;
  FTopSpeed: Integer;
protected
  FEngine: TEngine;
public
  constructor create (GPS: TGPSReceiver; color: string; Make: string;
    TopSpeed: Integer);
  destructor Destroy;
  procedure drive;
  property GPSReceiver: TGPSReceiver read FGPSReceiver write FGPSReceiver;
  property TopSpeed: Integer read FTopSpeed write FTopSpeed;
end;
```



```
procedure Register;
```

```
implementation
```

```
procedure Register;
```

```
begin
```

```
end;
```

```
{ * * * * * TWheel
```

```
constructor TWheel.create (size: Integer; TypeName: string; No: Integer);
```

```
begin
```

```
    FSize: = size;
```

```
    FTypeName: = TypeName;
```

```
    FNo: = No;
```

```
    check;
```

```
end;
```

```
procedure TWheel.check;
```

```
begin
```

```
    ShowMessage ('检查第' + IntToStr (FNo) + '个车轮。型号: ' + FTypeName + ' 大小: ' + IntToStr (FSize) );
```

```
end;
```

```
{ * * * * * TVehicle
```

```
procedure TVehicle.SlowDown;
```

```
begin
```

```
    ShowMessage ('正在减速 ...');
```

```
end;
```

```
procedure TVehicle.SpeedUp;
```

```
begin
```

```
    ShowMessage ('正在加速 ...');
```

```
end;
```

```
procedure TVehicle.Start;
```

```
begin
```

```
    ShowMessage ('车子开始启动');
```

```
end;
```

```
procedure TVehicle.Stop;
```

```
begin
```

```
    ShowMessage ('车子停下');
```

```
end;
```

```
{ * * * * * TBicycle
```

```
constructor TBicycle.create;
```

```
var
```

```
    i: Integer;
```

```
begin
```

```
    inherited create;
```

```
    FColor: = '白色';
```

```
    FMake: = '永久';
```

```
    FTopSpeed: = 20;
```

```
    FWheels: = TList.Create;
```

```
    for i: = 1 to 2 do
```

```

    FWheels.Add (TWheel.create (21,'B型自行车车轮',i) );
end;

destructor TBicycle.Destroy;
var
    i: Integer;
begin
    for i: = 1 to 2 do
        TWheel (FWheels.Items [i]) .Free;
    inherited;
end;

procedure TBicycle.ride;
begin
    start;
    speedUp;
    showMessage ('自行车行驶中 ... ');
    SlowDown;
    Stop;
end;

{ * * * * * TCar {
constructor TCar.create (GPS: TGPSReceiver; color: string; Make: string;
    TopSpeed: Integer );
var
    i: Integer;
begin
    inherited create;
    FEngine: = TEngine.create;
    //假设最高时速与发动机属性有逻辑关系
    FEngine.Fcapacity: = TopSpeed+1000;
    FEngine.Fpower: = TopSpeed-70;
    FColor: = color;
    FMake: = Make;
    FTopSpeed: = TopSpeed;
    FWheels: = TList.Create;
    ShowMessage (FColor + FMake + '车,最高时速: ' + IntToStr (FTopSpeed) );
    for i: = 1 to 4 do
        FWheels.Add (TWheel.create (36,'A型汽车车轮',i) );
    GPSReceiver: = GPS;
end;

destructor TCar.Destroy;
var
    i: Integer;
begin
    for i: = 1 to 4 do
        TWheel (FWheels.Items [i]) .Free;
    FEngine.Free;
    inherited;
end;

procedure TCar.drive;
begin

```

```

start;
if assigned (GPSReceiver) then GPSReceiver.Navigate;
FEngine.start;
speedUp;
showmessage ('汽车行驶中 ...');
SlowDown;
FEngine.stop;
Stop;
end;

{ * * * * * TGPSReceiver * * * * * }
procedure TGPSReceiver.Navigate;
begin
  showmessage ('使用 GPS 导航');
end;

{ * * * * * TEngine * * * * * }

procedure TEngine.start;
begin
  ShowMessage (inttostr (Fcapacity) + ' cc,' + inttostr (Fpower)
    + '匹马力的发动机发动了! ');
end;

procedure TEngine.stop;
begin
  ShowMessage ('发动机关闭了');
end;

end.

// - - - - -
// 汽车的聚合关系和组合关系示例程序 V3.0
// 用户界面单元 (frmDemo.pas): 该单元包含了操作演示界面
// 刘艺 2003/04/03
// - - - - -

unit frmDemo;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Demo, jpeg, ExtCtrls;

type
  TForm1 = class (TForm)
    btnRideBicycle: TButton;
    btnDriveCarA: TButton;
    ckbGPS: TCheckBox;
    btnDriveCarB: TButton;
    Image1: TImage;
    Button1: TButton;
    procedure btnDriveCarAClick (Sender: TObject);
    procedure btnRideBicycleClick (Sender: TObject);
  end;

```

```
    procedure ckbGPSClick (Sender: TObject);
    procedure btnDriveCarBClick (Sender: TObject);
    procedure Button1Click (Sender: TObject);
private
    FGPSReceiver: TGPSReceiver;
public

end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.btnDriveCarAClick (Sender: TObject);
var MyCar: TCar;
begin
    MyCar: = TCar.Create (FGPSReceiver, '黑色', '红旗', 200);
    try
        MyCar.drive;
    finally
        MyCar.Free;
    end;
end;

procedure TForm1.btnRideBicycleClick (Sender: TObject);
var Bicycle: TBicycle;
begin
    Bicycle: = TBicycle.Create;
    try
        Bicycle.ride;
    finally
        Bicycle.Free;
    end;
end;

procedure TForm1.ckbGPSClick (Sender: TObject);
begin
    if ckbGPS.Checked then
        FGPSReceiver: = TGPSReceiver.Create
    else
        FreeAndNil (FGPSReceiver);
    if assigned (FGPSReceiver) then showmessage ('导航系统开启 ...')
    else showmessage ('导航系统关闭 ...');
end;

procedure TForm1.btnDriveCarBClick (Sender: TObject);
var MyCar: TCar;
begin
    MyCar: = TCar.Create (FGPSReceiver, '红色', '夏利', 120);
    try
        MyCar.drive;
```

```

finally
    MyCar.Free;
end;
end;

procedure TForm1.Button1Click (Sender: TObject );
begin
    close;
end;

end.

```

运行程序，如图 3-19 所示。我们可以验证，只要 GPS 导航系统工作，就可以被所有 TCar 的实例对象使用。因为 TGPSReceiver 的实例对象的创建和销毁不是由 TCar 的实例对象负责的。它们之间的生命期没有必然的联系。而每个 TCar 的实例对象则分别有自己单独的发动机对象，这些发动机对象是由汽车对象创建和使用的，这就是说汽车部件对象是为汽车对象而存活的。

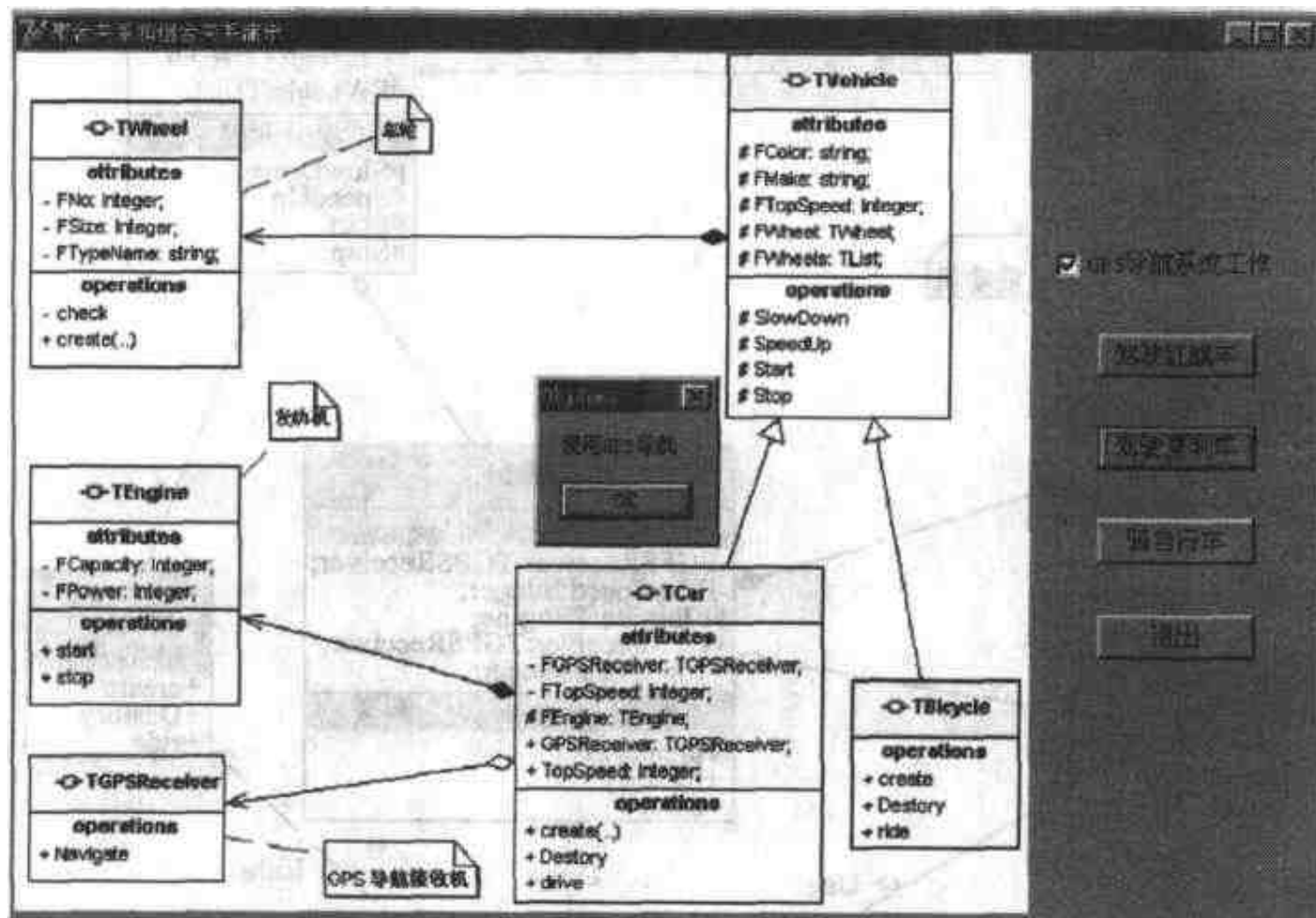


图 3-19 汽车的聚合关系和组合关系示例程序运行界面

3.3.4 依赖关系和合作关系

依赖（dependency）关系和合作（association）关系也是着眼于对象间互相作用的一种关系。依赖关系表示一个对象需要使用另一个对象，它依赖于另一个对象的定义，被依赖的对象是为依赖对象而存活的。合作关系表示一个对象的存活不受另一个对象的控制，这两个对象之间存在着平等自由的合作关系，合作对象和主动合作的对象分享它的生命期。可以把合作关系看做是一种被依赖对象生命期不受约束的弱依赖关系。从对象之间的生命期看，依赖关系有点类似组合关系，合作关系有点类似聚合关系。不同之处在于依赖关系和合作关系中的对象是以局部

变量和方法参数的形式存在，而组合关系和聚合关系中的变量是以数据成员或属性的形式存在。

下面我来举例说明。一个人可以骑自行车、开汽车和使用 GPS 导航接收机，如图 3-20 所示。在这个例子中，TPerson 类的对象与 TCar 类的对象和 TBicycle 类的对象存在依赖关系。在 UML 类图中，我们用虚线箭头指出了它们之间的依赖关系。TPerson 类依赖于 TCar 类和 TBicycle 类的定义，因为它分别引用了 TCar 类和 TBicycle 类的实例变量作为 Drive 和 Ride 方法的参数。需要说明的是汽车对象和自行车对象都是为人对象的 Drive 和 Ride 方法存活的，在示例程序 3-15 中，从骑自行车按钮事件 (TForm1.btnRideBicycleClick) 和驾驶汽车按钮事件 (TForm1.btnDriveCarBClick 和 TForm1.btnDriveCarBClick) 的代码中我们可以看到对象的使用情况及其生命期。

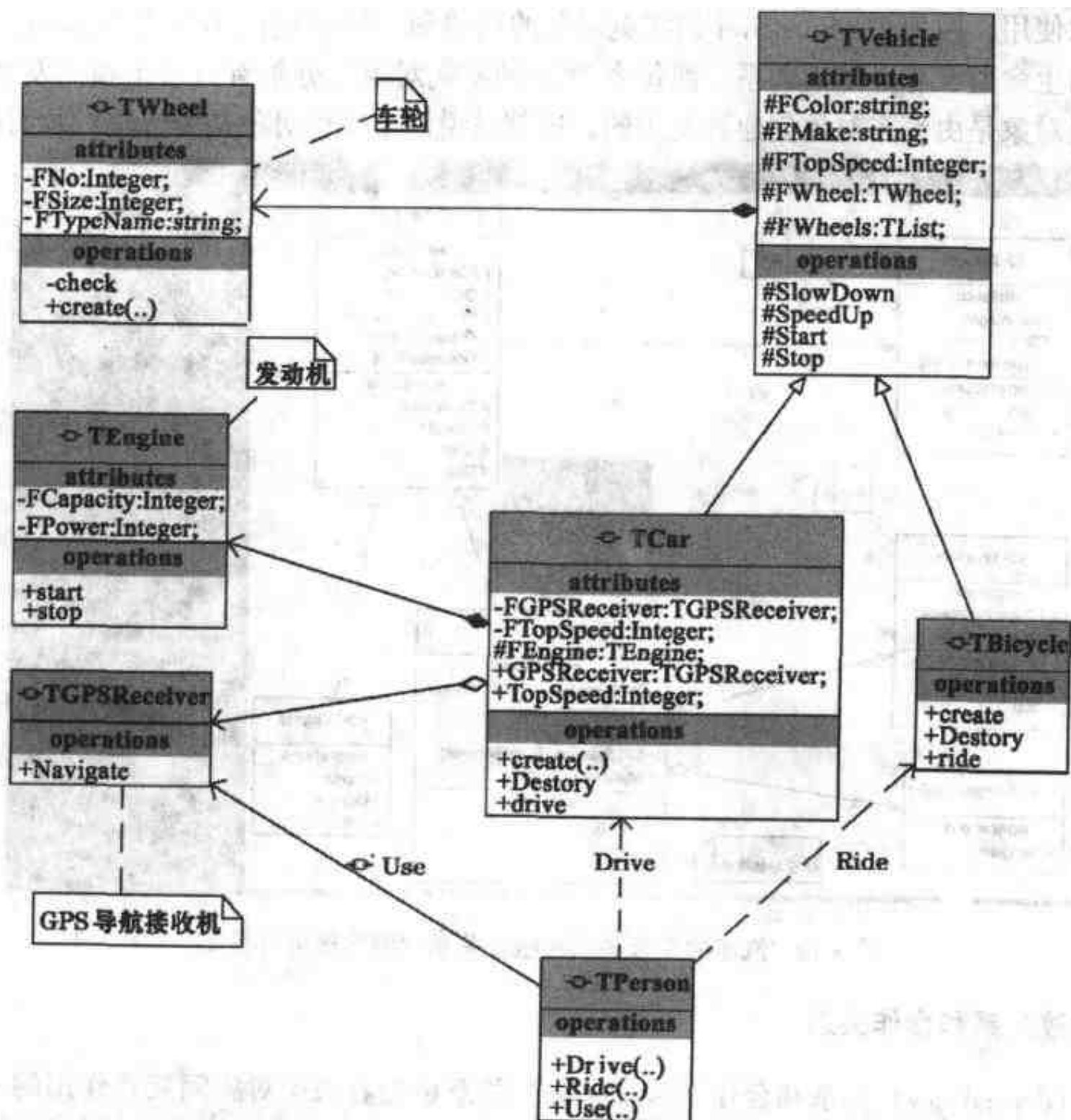


图 3-20 依赖关系和合作关系示例程序 UML 图

不同的是 TGPSReceiver 类的对象和 TPerson 类的对象之间是合作关系，在 UML 类图中，我们用实线箭头指出了它们之间的合作关系。TGPSReceiver 类的对象和 TPerson 类的对象都是作为 TForm1 实例对象的数据成员，它们的生命期没有相互控制的关系。前者由 TForm1.ckbGPSClick

事件创建, 后者在 TForm1.FormCreate 事件中创建, 它们创建和销毁的时间没有联系。所以 TGPSReceiver 实例对象和 TPerson 实例对象之间是合作关系, 这种合作具体体现在 TPerson 的 Use (GPSReceiver: TGPSReceiver) 方法中。

示例程序 3-15 依赖关系和合作关系示例程序

```
// - - - - -
// 汽车的依赖关系和合作关系示例程序 v3.0
// 业务逻辑单元 (Demo.pas): 该单元包含了汽车及其零部件对象
// 刘艺 2003/04/03
// ~ - - - - -

unit Demo;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TGPSReceiver = class (TObject)
  public
    procedure Navigate;
  end;

  TEngine = class (TObject)
  private
    FCapacity: Integer;
    FPower: Integer;
  public
    procedure start;
    procedure stop;
  end;

  TWheel = class (TObject)
  private
    FNo: Integer;
    FSize: Integer;
    FTypeName: string;
    procedure check;
  public
    constructor create (size: Integer; TypeName: string; No: Integer);
  end;

  TVehicle = class (TObject)
  protected
    FColor: string;
    FMake: string;
    FTopSpeed: Integer;
    FWheel: TWheel;
    FWheels: TList;
    procedure SlowDown;
```

```

    procedure SpeedUp;
    procedure Start;
    procedure Stop;
end;

TBicycle = class (TVehicle)
public
    constructor create;
    destructor Destory;
    procedure ride;
end;

TCar = class (TVehicle)
private
    FGPSReceiver: TGPSReceiver;
    FTopSpeed: Integer;
protected
    FEngine: TEngine;
public
    constructor create (GPS: TGPSReceiver; color: string; Make: string;
        TopSpeed: Integer);
    destructor Destory;
    procedure drive;
    property GPSReceiver: TGPSReceiver read FGPSReceiver write FGPSReceiver;
    property TopSpeed: Integer read FTopSpeed write FTopSpeed;
end;

TPerson = class (TObject)
public
    procedure Drive (car: TCar);
    procedure Ride (Bicycle: TBicycle);
    procedure Use (GPSReceiver: TGPSReceiver);
end;

procedure Register;

implementation

procedure Register;
begin
end;

{ * * * * * TEngine * * * * * }
procedure TEngine.start;
begin
    ShowMessage (inttostr (Fcapacity) + 'cc,' + inttostr (Fpower) +
        '匹马力的发动机发动了!');
end;

procedure TEngine.stop;
begin
    ShowMessage ('发动机关闭了');
end;

```

```

{ * * * * * TWheel * * * * * }
constructor TWheel.create (size: Integer; TypeName: string; No: Integer);
begin
    FSize: = size;
    FTypeName: = TypeName;
    FNo: = No;
    check;
end;

procedure TWheel.check;
begin
    ShowMessage ('检查第' + IntToStr (FNo) + '个车轮。型号: ' +
        FTypeName + ' 大小: ' + IntToStr (FSize) );
end;

{ * * * * * TVehicle * * * * * }
procedure TVehicle.SlowDown;
begin
    ShowMessage ('正在减速 ... ');
end;

procedure TVehicle.SpeedUp;
begin
    ShowMessage ('正在加速 ... ');
end;

procedure TVehicle.Start;
begin
    ShowMessage ('车子开始启动');
end;

procedure TVehicle.Stop;
begin
    ShowMessage ('车子停下');
end;

{ * * * * * TBicycle * * * * * }
constructor TBicycle.create;
var
    i: Integer;
begin
    inherited create;
    FColor: = '白色';
    FMake: = '永久';
    FTopSpeed: = 20;
    FWheels: = TList.Create;
    for i: = 1 to 2 do
        FWheels.Add (TWheel.create (21, 'B 型自行车车轮', i) );
    end;

destructor TBicycle.Destroy;
var
    i: Integer;
begin

```

```

    for i: =1 to 2 do
        TWheel (FWheels.Items [i]) .Free;
    inherited;
end;

procedure TBicycle.ride;
begin
    start;
    speedUp;
    showmessage ('自行车行驶中 ...');
    SlowDown;
    Stop;
end;

{ * * * * * TCar * * * * * }
constructor TCar.create (GPS: TGPSReceiver; color: string; Make: string;
    TopSpeed: Integer);
var
    i: Integer;
begin
    inherited create;
    FEngine: = TEngine.create;
    //假设最高时速与发动机属性有逻辑关系
    FEngine.Fcapacity: = TopSpeed+1000;
    FEngine.Fpower: = TopSpeed-70;
    FColor: = color;
    FMake: = Make;
    FTopSpeed: = TopSpeed;
    FWheels: = TList.Create;
    ShowMessage (FColor + FMake + '车,最高时速: ' + IntToStr (FTopSpeed) );
    for i: =1 to 4 do
        FWheels.Add (TWheel.create (36, 'A型汽车车轮', i) );
    GPSReceiver: = GPS;
end;

destructor TCar.Destroy;
var
    i: Integer;
begin
    for i: =1 to 4 do
        TWheel (FWheels.Items [i]) .Free;
    FEngine.Free;
    inherited;
end;

procedure TCar.drive;
begin
    start;
    if assigned (GPSReceiver) then GPSReceiver.Navigate;
    FEngine.start;
    speedUp;
    showmessage ('汽车行驶中 ...');
    SlowDown;
    FEngine.stop;
end;

```

```

    Stop;
end;

}*****TPerson}
procedure TPerson.Drive (car: TCar);
begin
    car.drive;
end;

procedure TPerson.Ride (Bicycle: TBicycle);
begin
    Bicycle.ride;
end;

procedure TPerson.Use (GPSReceiver: TGPSReceiver);
begin
    GPSReceiver.Navigate;
end;

}*****TGPSReceiver}
procedure TGPSReceiver.Navigate;
begin
    showmessage ('使用 GPS 导航');
end;

end.

//-----
// 汽车的依赖关系和合作关系示例程序 v3.0
// 用户界面单元 (frmDemo.pas)
//          刘艺 2003/04/03
//-----
unit frmDemo;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, Demo;

type
    TForm1 = class (TForm)
        btnRideBicycle: TButton;
        btnDriveCarA: TButton;
        ckbGPS: TCheckBox;
        btnDriveCarB: TButton;
        btnUseGPS: TButton;
        procedure btnDriveCarAClick (Sender: TObject);
        procedure btnRideBicycleClick (Sender: TObject);
        procedure ckbGPSClick (Sender: TObject);
        procedure btnDriveCarBClick (Sender: TObject);
        procedure FormCreate (Sender: TObject);
    end;

```

```
    procedure btnUseGPSClick (Sender: TObject);
    procedure FormDestroy (Sender: TObject);
private
    FGPSReceiver: TGPSReceiver;
    APerson: TPerson;
public

end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.btnDriveCarAClick (Sender: TObject);
var MyCar: TCar;
begin
    MyCar: = TCar.Create (FGPSReceiver, '黑色', '红旗', 200);
    APerson.drive (MyCar);
    MyCar.Free;
end;

procedure TForm1.btnRideBicycleClick (Sender: TObject);
var Bicycle: TBicycle;
begin
    Bicycle: = TBicycle.Create;
    APerson.Ride (Bicycle);
    Bicycle.Free;
end;

procedure TForm1.ckbGPSClick (Sender: TObject);
begin
    if ckbGPS.Checked then
        FGPSReceiver: = TGPSReceiver.Create
    else
        FreeAndNil (FGPSReceiver);
    if assigned (FGPSReceiver) then showmessage ('导航系统开启 ...')
    else showmessage ('导航系统关闭 ...');
end;

procedure TForm1.btnDriveCarBClick (Sender: TObject);
var MyCar: TCar;
begin
    MyCar: = TCar.Create (FGPSReceiver, '红色', '夏利', 120);
    APerson.drive (MyCar);
    MyCar.Free;
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
    APerson: = TPerson.Create;
```

```
end;  
  
procedure TForm1.btnUseGPSClick (Sender: TObject);  
begin  
    if assigned (FGPSReceiver) then APerson.use (FGPSReceiver)  
    else Showmessage ('GPS 系统没有工作,无法使用! ');  
end;  
  
procedure TForm1.FormDestroy (Sender: TObject);  
begin  
    APerson.Free;  
    if assigned (FGPSReceiver) then FGPSReceiver.Free;  
end;  
  
end.
```

从示例程序中我们可以进一步理解合作关系和依赖关系的差别。在合作关系中,合作对象与主动合作的对象分享生命期,所有合作对象是平等的关系,它们的生命期要尽量一样长,以保证合作成功。在依赖关系中,可以在依赖该对象的过程中创建和销毁所依赖的对象,使被依赖对象的生命期尽可能短。特别是在多层分布式系统中,设计好对象的合作关系和依赖关系意义非常大。比如,某个客户对象请求一个位于中间层的业务逻辑对象提供服务时,它对这个业务对象存在依赖关系。于是客户对象创建这个业务对象的实例,使用完该项服务后需要立即销毁该对象实例。因为在分布式环境中,可能需要请求该项服务的客户对象很多,而网络 and 系统的资源十分有限,所以,应该尽可能缩短所依赖对象的生命期,以提高分布式系统的运行效率。但是,对于中间层的协调控制对象来说,它的生命期可能就要和客户对象一样长,因为它是它帮助客户对象协调和使用中间层的各个业务逻辑对象的,所以协调控制对象和客户对象存在着合作关系。

第4章 使用对象

4.1 应用程序和界面对象

4.1.1 Windows 应用程序和 Application 对象

我们知道 Windows 应用程序是基于 Windows 窗体的。通过 Windows 程序的窗体，可以完成 Windows 消息循环，保证程序的正常运行。但是如果使用纯 API 编写 Windows 应用程序、创建窗体、处理消息，将会是非常麻烦的。好在 Delphi 提供了 Application 对象，可以帮助我们定义一个应用程序的特性和行为，分发和处理 Windows 消息，使得创建 Windows 应用程序简单易行。由 Application 对象创建的应用程序实际上是一个没有大小、不可见的窗体，它可以出现在 Windows 任务栏上。显然它不是程序员自己定义的 Main Form，因为它显示的标题来自于 TApplication.Title 的值，而决非 TForm1.Caption 的值。应该说，程序中所有的 Form，都是它的子窗体。

当创建一个默认的应用程序时，我们会看到以下代码：

```
program Project1;  
  
uses  
    Forms,  
    Unit1 in 'Unit1.pas' {Form1};  
  
{$R *.res}  
  
begin  
    Application.Initialize;  
    Application.CreateForm (TForm1, Form1);  
    Application.Run;  
end.
```

这是一个应用程序不可缺少，但又非常简洁的几行代码。通过这几行代码，Application 对象完成了程序初始化、创建主窗体、运行程序等重要功能。另外，Application 对象还为程序提供了许多有用的功能，例如终止程序：

```
Application.Terminate;
```

例如弹出提示信息对话框：

```
Application.MessageBox ('这是应用程序的提示信息。', '注意', [smbOK] );
```

例如捕捉和显示应用程序级的错误：

```
procedure TForm1.FormCreate (Sender; TObject);  
begin  
    Application.OnException := AppException;  
end;
```

```
procedure TForm1.AppException (Sender: TObject; E: Exception);  
begin  
    Application.ShowException (E);  
    Application.Terminate;  
end;
```

Application 对象是在 Forms 单元中声明的 TApplication 类型全局变量，可以在程序中直接使用。

参见 关于 TApplication 类的更多细节，我会在后面第 9 章中进一步介绍。

4.1.2 窗体和对话框

窗体和对话框是 Windows 编程中一类非常重要的对象。窗体继承自 TComponent，具有组件可视化设计的特征，支持流。它是 TWinControl 子类的派生类，拥有标准的 Windows 窗体句柄、输入焦点和自己的消息队列管理函数。在 Delphi 对象浏览器中，可以查看到窗体的继承关系和方法属性，如图 4-1 所示。

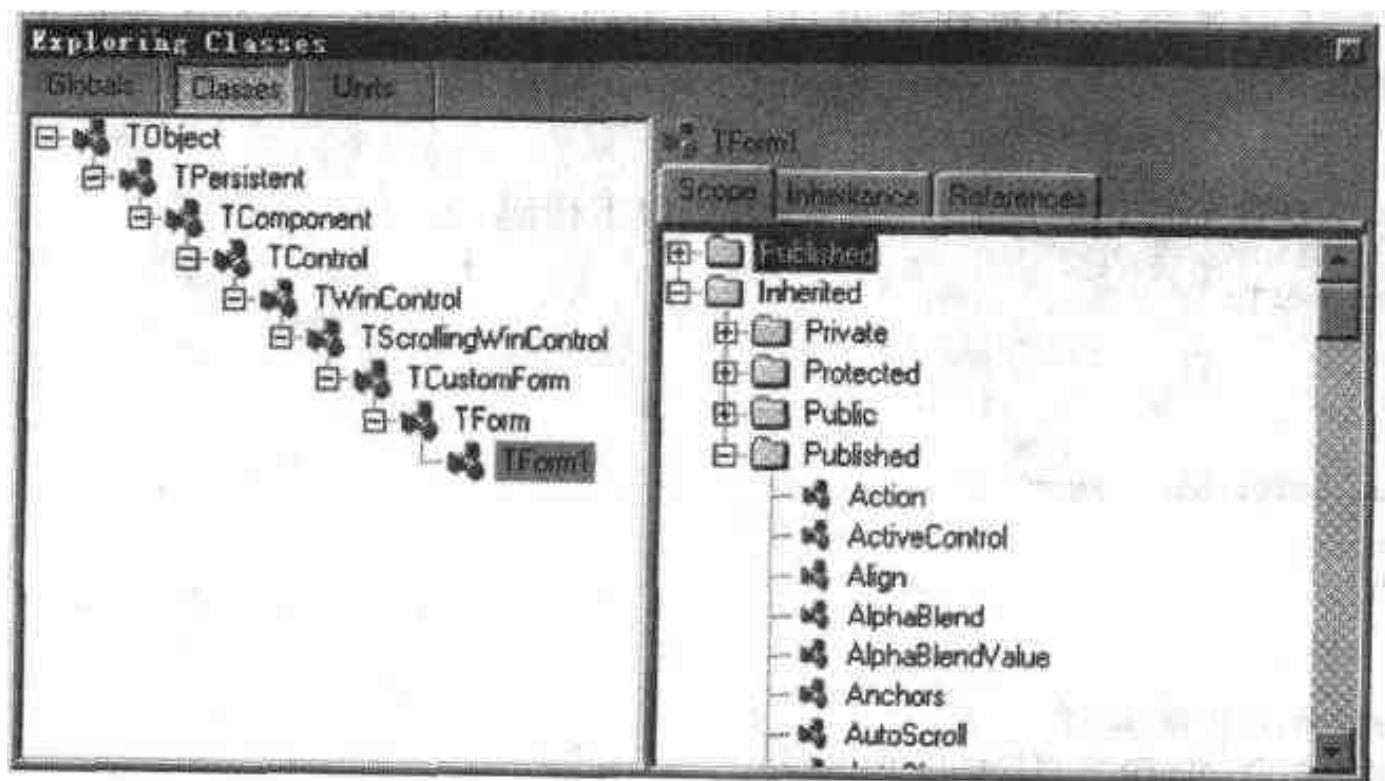


图 4-1 在对象浏览器中查看窗体的继承关系

窗体实际上是一个界面容器，它可以包含其他的组件对象。通过 Delphi 优秀的 RAD 开发功能，我们可以在窗体上可视化地完成组件对象的布局，设计出美观的图形用户界面。当窗体上组件比较多时或相互遮盖时，我们可以使用 Object TreeView 来查看和管理窗体及其所属组件。图 4-2 就是一个设计中的窗体和 Object TreeView，我们可以观察到窗体作为容器所包含的组件。

在 Windows 应用程序中，用户界面可分为 SDI（单文档界面）和 MDI（多文档界面），虽然 MDI 可以在一个主窗体中同时打开多个子窗体，方便用户操作，但以微软为代表的主流产品则更倾向于使用简单的 SDI 窗体界面。实际上，在 Delphi 中，开发 SDI 和 MDI 窗体界面都很方便。

当我们把窗体添加到应用程序中时，Delphi 会自动将一行代码加入到项目原文件（.DPR）

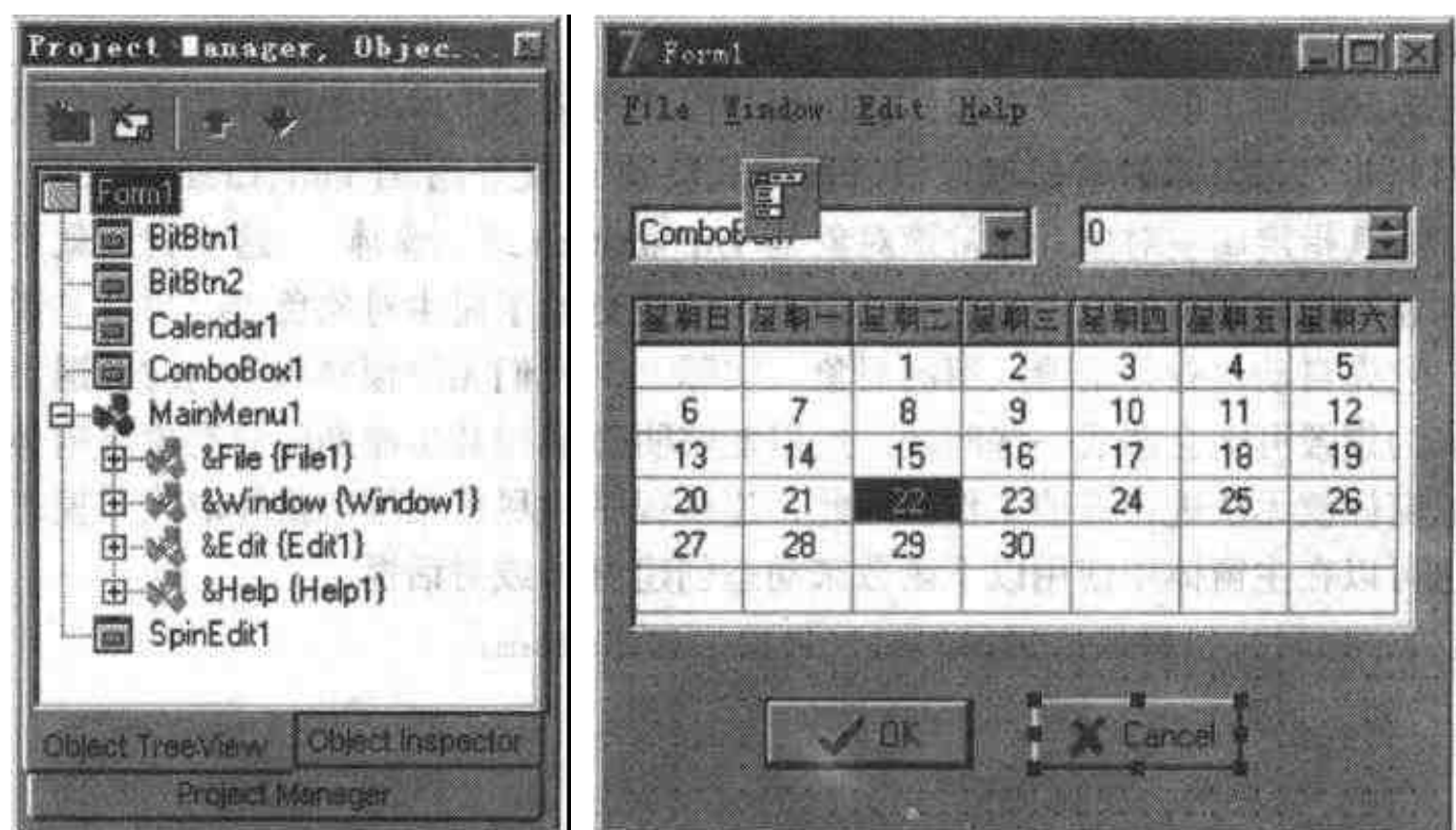


图 4-2 在 Object TreeView 上可以观察到窗体作为容器所包含的组件

中，以便在程序启动时自动创建窗体：

```
Application.CreateForm (TForm1, Form1);
```

对于没有完全习惯于面向对象编程的程序员来说，这可以使得添加和使用窗体更为简单。但这并不意味着 Delphi 没有强大的面向对象编程能力，实际上，Delphi 的过人之处在于它既可以让新手快速入门，又可以让高手创建复杂的应用程序。这里我不打算讨论窗体和对话框的使用方法，我讨论的重点在于如何利用面向对象编程思想来优化窗体和对话框编程中的代码实现。

在使用 Delphi 开发窗体时，我们要尽量避免让 Delphi 自动创建它们，因为每一个创建好的窗体都要消耗一定的系统资源，尽管有些窗体创建后我们暂时还用不到（看不到）它们。反之，如果任由 Delphi 自动创建所有的窗体，在启动时将会影响程序的加载速度，让用户为没用到的或很少用到的窗体付出性能方面的代价。

在一个复杂的系统中，我们可以通过动态创建窗体来节约系统资源。

比较常用的方法是调用窗体的构造方法来显式创建窗体。

```
Form1 := TForm1.Create (nil);  
try  
    if (Form1.ShowModal = mrOK) then  
        //必要的程序编码写在这里  
    finally  
        Form1.Free;  
    end;
```

或

```
TForm1.Create (self).show;
```

前一种方法适用于创建模态窗体。模态窗体是独占使用的，即打开模态窗体时，其他窗体无法同时工作。所以关闭模态窗体时要记住销毁它。注意，这里不用 `TForm1.Create (self)`，为

什么？不清楚的读者可参见 3.2.4 节。

后一种方法适用于创建非模态窗体。由于非模态窗体不排斥其他窗体，可以和其他窗体共同工作，因此非模态窗体的生命期比模态窗体长。我们最好用 `TForm1.Create (self)` 来创建非模态窗体，为其指定属主对象（通常该对象是 `TApplication` 或主窗体）。这样我们就不必考虑在何处销毁创建的非模态窗体，因为它的销毁工作已经交给了属主对象负责，也就是说当属主对象销毁时，它先自动检查并销毁从属的对象。实际上，我们无法预料用户何时会退出非模态窗体，因而人为销毁可能会造成一些问题。所以能够明确掌握其生命期的只有模态窗体，因为用户不退出该窗体就无法进行其他工作。因此，建议程序中尽量用模态窗体取代非模态窗体。

当然也可以在主窗体中使用以下函数来动态创建窗体或对话框：

```
function TForm1.createdform (fromclass: TFromclass): TForm;
var
    lForm: TForm;
begin
    lForm := fromclass.Create (self);
    //如果需要统一管理窗体,这里可以添加统一控制代码
    result := lForm;
end;
```

注意，这里使用的参数类型是 `TFromclass`，`fromclass` 传递的是类引用而不是对象引用。但函数的返回值却是对象引用。`TFromclass` 的定义如下：

```
TFromclass = class of TForm;
```

显然，我们使用的窗体和对话框（窗体类型）都是 `TForm` 的派生类，所以可以如此处理。这也叫做向上转型，第 5 章中我会详细介绍转型技术。

因此我们可以通过这个 `createdform` 方法创建窗体，比如创建 `TForm2` 窗体：

```
Form2 := (createdform (TForm2) as TForm2);
```

或

```
Form2 := TForm2 (createdform (TForm2));
```

由于 `createdform` 方法创建窗体返回的是 `TForm` 类型，因此这里又使用了一次向下转型，确保得到的对象引用是 `TForm2` 的实例对象。示例程序 4-1 给出了 `createdform` 方法动态创建窗体的完整源代码。

示例程序 4-1 `createdform` 方法动态创建窗体

```
unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, Unit2, Unit3, Unit4;

type
    TFromclass = class of TForm;
```

```

TForm1 = class (TForm)
  Button1: TButton;
  Memo1: TMemo;
  Button2: TButton;
  Button3: TButton;
  Button4: TButton;
  procedure Button1Click (Sender: TObject);
  procedure Button2Click (Sender: TObject);
  procedure Button3Click (Sender: TObject);
  procedure Button4Click (Sender: TObject);
private
  Fform2: TForm2;
  Fform3: TForm3;
  function createdform (fromclass: TFromclass): TForm;
public
  {Public declarations }
end;

implementation

{$R *.dfm}

function TForm1.createdform (fromclass: TFromclass): TForm;
var
  lForm: TForm;
  tmpstr: string;
begin
  lForm := fromclass.Create (self);
  //这里可以添加统一控制代码
  //记录窗体创建的时间
  tmpstr := lForm.Caption + ' [' + formatdatetime ('hh: mm: ss', now) + '] 创建!';
  memo1.Lines.Add (tmpstr);
  result := lForm;
end;

procedure TForm1.Button1Click (Sender: TObject);
begin
  //非模态窗体只需创建一次,该窗体关闭时只是隐藏起来,并没有消失。
  if (Fform2 = nil) then
    Fform2 := TForm2 (createdform (Tform2));
  //对于已经创建的非模态窗体 (包括隐藏起来的) 通过 Show 来调用并显示。
  Fform2.Show;
end;

procedure TForm1.Button2Click (Sender: TObject);
var
  OKHelpBottomDlg: TOKHelpBottomDlg;
begin
  //对话框是模态窗体,这里演示模态窗体的使用
  OKHelpBottomDlg := TOKHelpBottomDlg (createdform (TOKHelpBottomDlg));
  try
    if (OKHelpBottomDlg.ShowModal = mrOK)
    then showmessage ('OK! 退出对话框!'); //这里可添加必要的代码
  finally

```

```
    OKHelpBottomDlg.free;  
end;  
end;  
  
procedure TForm1.Button3Click (Sender: TObject);  
begin  
    if (Fform3 = nil) then  
        Fform3: = (createdform (Tform3) as TForm3);  
        Fform3.Show;  
    end;  
  
procedure TForm1.Button4Click (Sender: TObject);  
begin  
    close;  
end;  
  
end.
```

示例程序 4-1 给出的 Unit1 单元是一个主窗体单元，通过这个主窗体上的三个按钮，我们可以调用子窗体 Form2、Form3（非模态窗体）以及对话框 OKHelpBottomDlg（模态窗体）。主窗体上的 Memo 组件记录并显示了窗体创建的时间。运行该程序时，对于非模态窗体而言该窗体只是在首次调用时创建。因为关闭（close）窗体只意味着隐藏窗体（hide），下次调用时只需“show”一下，而不必重新创建。但是，对于非模态窗体而言，其生命期是可以管理的，所以在每次调用时创建，用完后销毁。

主窗体运行界面如图 4-3 所示。从窗体创建的时间记录上，我们可以直观地看到对话框可以多次创建。

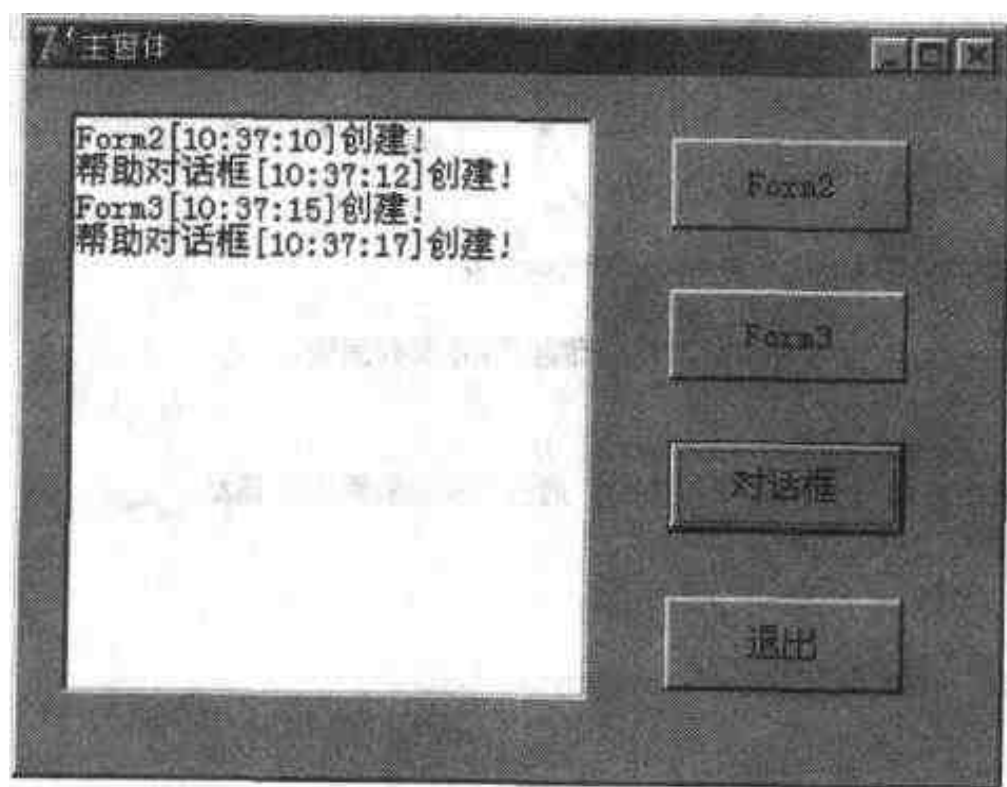


图 4-3 主窗体运行界面

手工动态创建窗体后，要记住清除 program 中由 Delphi 自动生成的创建窗体的代码，这些代码是以这样的形式存在的：

```
Application.CreateForm (TFormXXX, FormXXX);
```


而且 Delphi 还在每个窗体单元自动生成了一个该窗体的全局变量 FormXXX, 通常在该单元的 implementation 限定符之前出现:

```
Var  
FormXXX: TFormXXX;
```

许多程序员通过 Use 不同的窗体单元, 很随便地使用这些全局变量, 破坏了窗体作为一个类的封装性。这不是面向对象的思维, 是编程中十分忌讳的坏习惯。我在第 7 章“研究封装”中会深入探讨这个问题。

由于我在示例程序 4-1 中没有使用全局变量 Form1, 并编程实现了子窗体的动态生成。所以 program 中的代码修改如下:

```
program Project2;  
  
uses  
  Forms,  
  Unit1 in 'Unit1.pas' {Form1},  
  Unit2 in 'Unit2.pas' {Form2},  
  {OKBottomDlg}  
  OKCANCL1 in  
    'C:\PROGRAM FILES\BORLAND\DELPHI7\ObjRepos\OKCANCL1.pas',  
  Unit3 in 'Unit3.pas' {OKHelpBottomDlg},  
  Unit4 in 'Unit4.pas' {Form3};  
  
{$R *.res}  
  
begin  
  Application.Initialize;  
  TForm1.create (Application) .ShowModal;  
  //Application.CreateForm (TForm1, Form1);  
  Application.Run;  
end.
```

读者可以在随书光盘中找到该应用程序的全部单元文件, 对于其他的窗体单元, 我也删除了 Delphi 为这些窗体单元自动生成的全局变量 FormXXX。不使用跨单元、跨类的全局变量可以起到去耦的效果。

为便于用户操作, 对于一些提示信息和较简单的交互输入我们可以使用信息框和输入框来实现, 以便加快程序开发速度。这里将它们统称为提示框。

Delphi 中的信息框与输入框函数有: Showmessage、ShowmessagePos、MessageDlg、MessageDlgPos、InputBox 和 InputQuery 等。这些函数都是以模态方式显示, 一旦打开这个信息框或输入框, 就必须在这个框上工作, 直到关闭它为止。

4.1.3 界面对象和 UI 框架

许多初学编程的朋友认为在 Delphi 中创建一个应用程序是一件非常简单的事情。通过 File|New|Application 菜单, Delphi 就可自动创建一个最简单的应用程序, 它包括了一个空白的主窗体。通过拖放控件、编写事件代码, 一般初学者都能很快入门, 写出一个有模有样的应用程序。如果编程仅仅是停留在这个层面上, 那么选择 Delphi 和选择 VB 实在没有太多的差别, 程

程序员水平也由使用的控件功能所决定。但实际上, Delphi 作为一种强大的面向对象开发工具, 在这种“RAD+可视化”的编程背后, 却深藏了许多复杂的对象机制。

如果从面向对象的角度来看, 其实应用程序 (Application) 可以看成是对象的集合, 如图 4-4 所示。所谓应用程序就是一个 Application 对象, 它包含了一些 Form 对象, 作为用户交互的界面。Form 对象本身是一种容器对象, 程序员在其中放置按钮和编辑框等控件处理具体的操作。类似的容器对象还有 Frame、DataModule 等, 用于对控件的集中管理。这些 Application、Form、Button 等对象都是派生自 TComponent 类, 所以它们都是组件对象。当然光有这些组件对象还不够用, 因为一个应用程序还有自己的业务逻辑和特定算法。所以除了 VCL 对象外, 在程序中还会有程序员自己编写的一些对象, 用于业务逻辑和特定算法, 比如: 客户对象、账单对象。

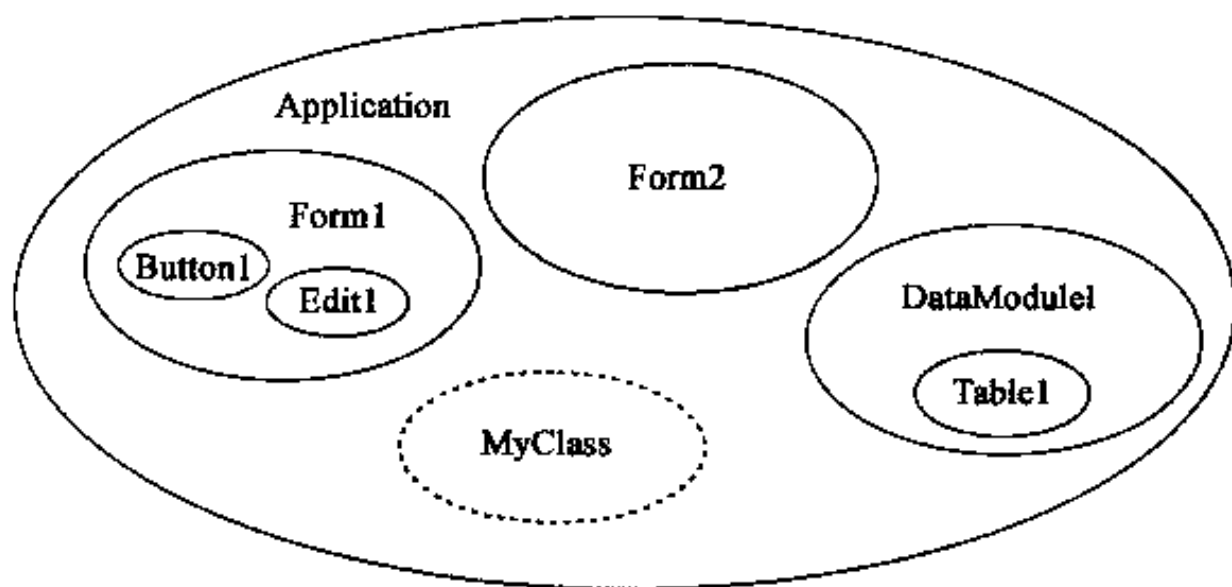


图 4-4 应用程序可以看成是对象的集合

尽管不同的应用程序的业务需求千差万别, 但几乎所有的 Windows 应用程序都有相似的界面和结构。这些用户界面 (UI) 包括: 主界面、对话框、菜单、按钮等已经成为业界标准的元素, Delphi 将这些元素封装成通用的可视化界面对象, 极大地方便了编程。在组织信息方面, 好的应用程序应该包括命令区、导航区、工作区等布局要素, 如图 4-5 所示。命令区由菜单、工具条组成, 用于用户操作命令的实现。导航区通常由树、列表框等控件提供信息快速分类或定位功能, 工作区可以浏览和处理信息。微软的资源管理器可以算得上是最经典的范例。

下面我就以图 4-5 所示的图书管理程序为例, 介绍一个标准的应用程序创建过程。这里重点介绍与界面设计有关的内容, 该示例程序的其他内容将在本章的后续章节中介绍。

这个图书管理程序是一个最简单的数据库应用程序, 使用了一个 Access 数据库的 mybook 表, 数据字段如图 4-6 所示。主界面提供了图书信息的浏览功能。导航区采用的是树的结构, 以便按照不同的分类提供信息导航功能。工作区使用了 ListView 组件, 可以有大图标、小图标、列表和详细信息 4 种方式组织信息, 方便地完成信息的显示。命令区有菜单和工具条, 符合 Windows 应用程序的习惯。

好的界面, 应该有一个功能区域划分明显的界面布局。另外还要不受用户运行程序时随意调整界面的影响。为此通常我们使用 panel 和 Splitter 完成布局设计, 这样既可以保持布局区域

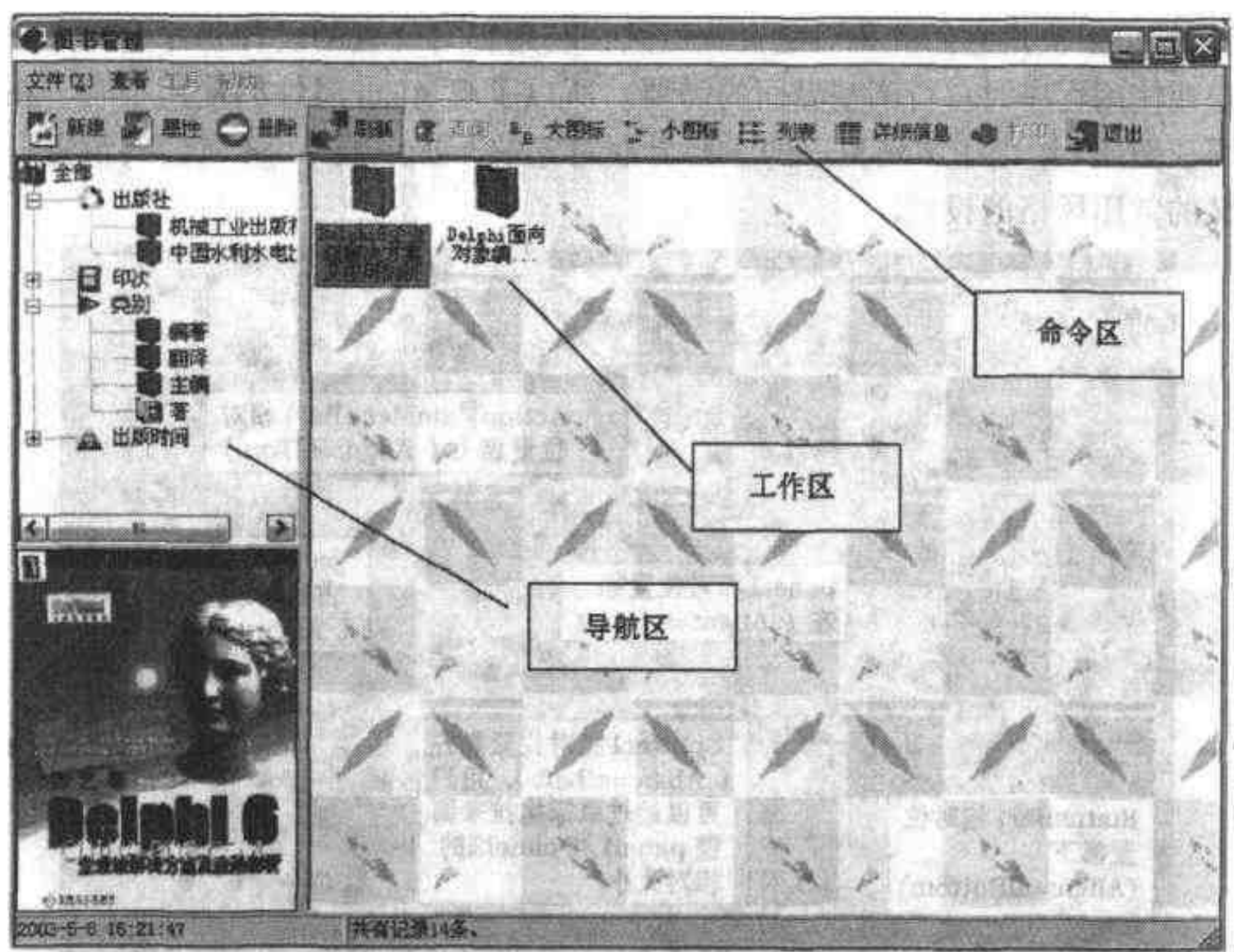


图 4-5 应用程序应该包括命令区、导航区、工作区等布局要素

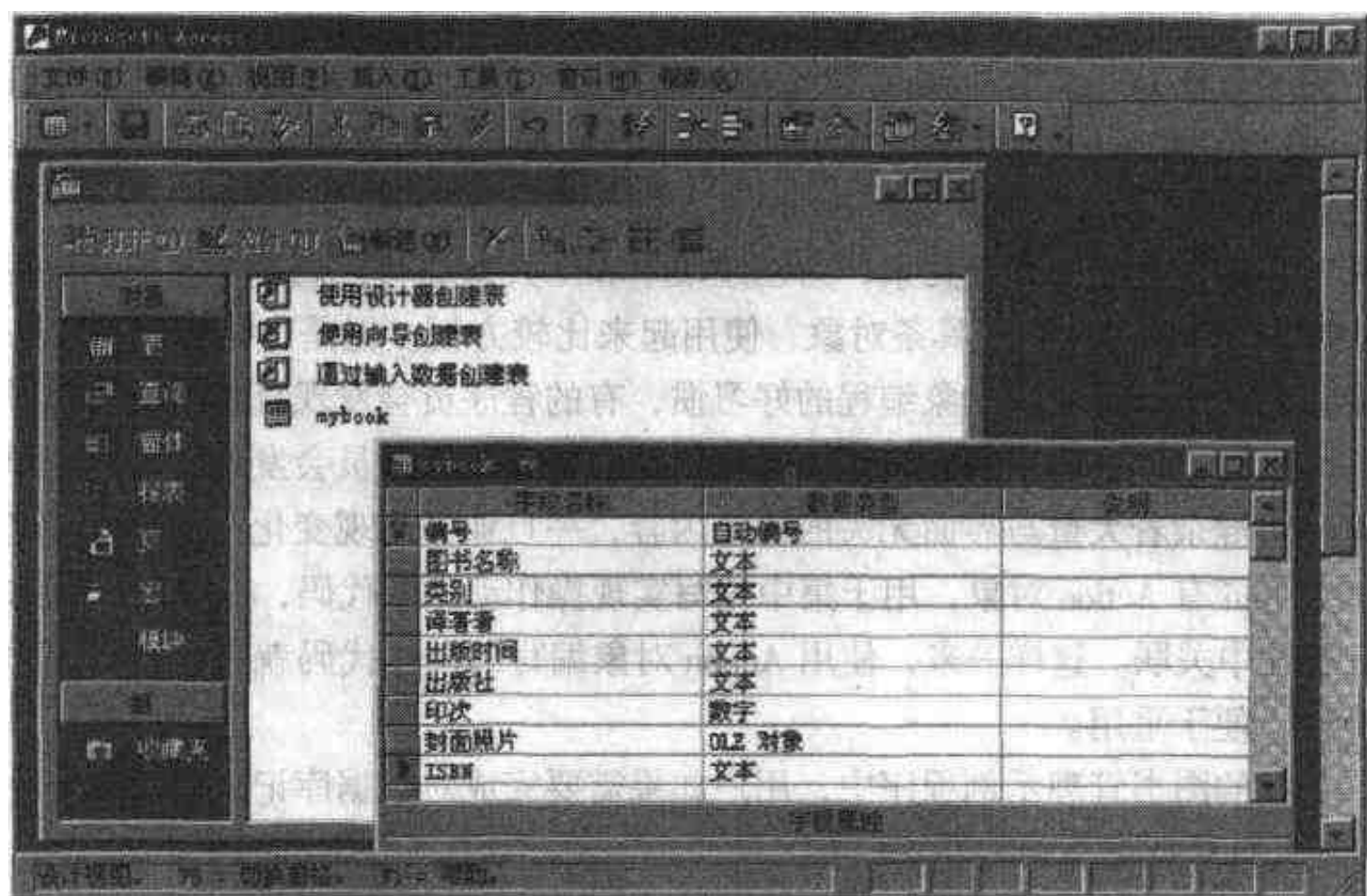


图 4-6 图书管理程序使用了 Access 数据库的 mybook 表

相对位置的稳定，也可以方便用户自己随意调整区域的相对大小。panel 对象本身就是一个容器，可以在其中放置其他的界面对象。Delphi 的可视化组件还有一个 Align 属性。通过界面对

象的 Align 属性可以决定它们的相对位置。显然, 利用相对位置的设计要比固定大小的设计更能满足用户在使用程序时自行调整界面的需要。所以了解 Align 属性的使用技巧, 在界面设计中十分重要。图 4-7 给出了示例程序的主要界面对象的布局以及 Align 属性的设置, 这是一个有指导意义的常用风格的设置。

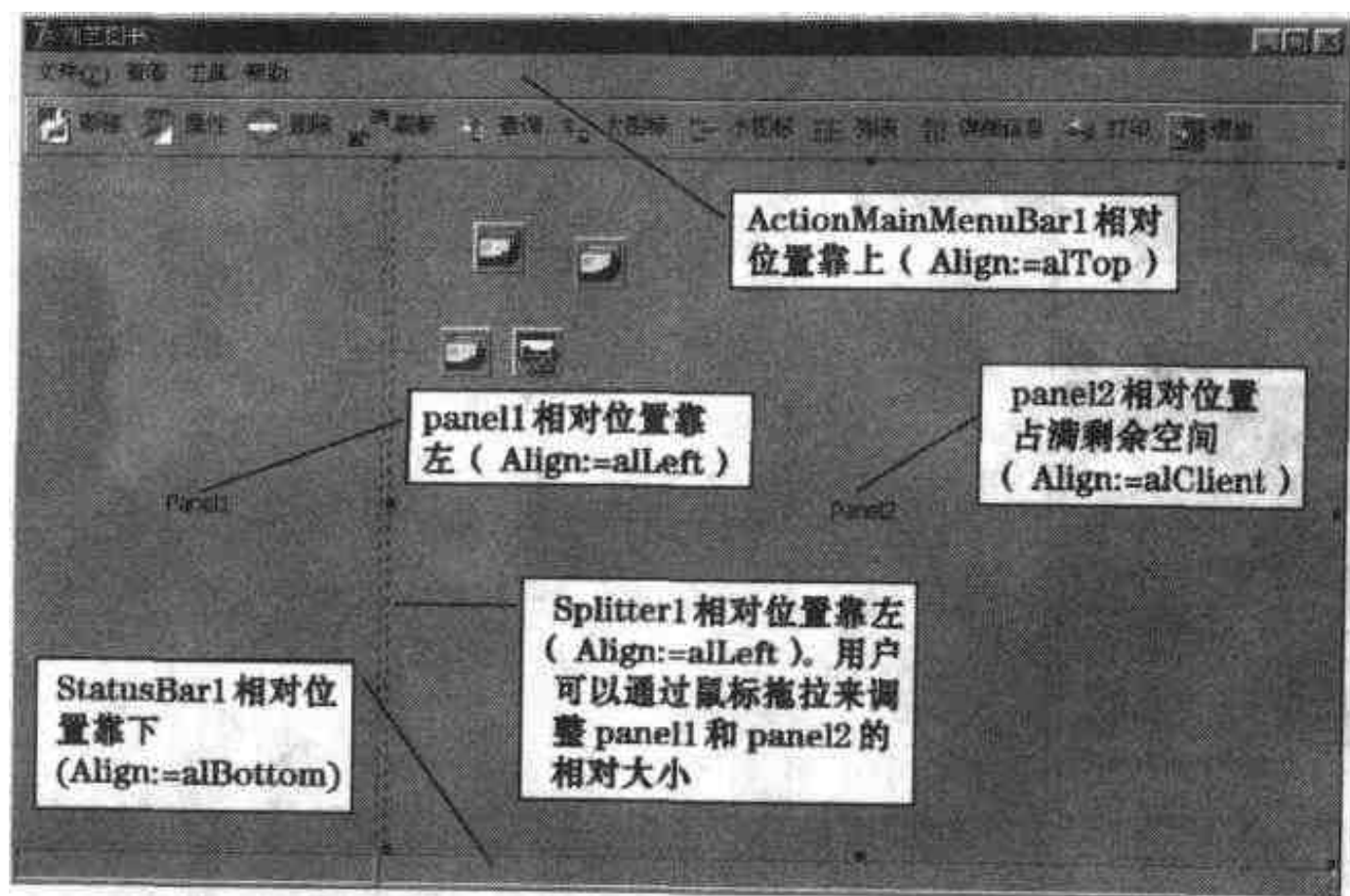


图 4-7 界面的布局设计——组件的 Align 属性决定其相对位置

值得一提的是在 Delphi 中, panel 和 Splitter 的配合十分巧妙, 示例程序中用户可以通过鼠标拖拉 Splitter1 来调整 panel1 和 panel2 的相对大小, 但在程序中却不需要写一行代码。这就得益于它们之间的 Align 属性设置。

菜单和工具条是重要的界面元素, 为创建菜单和工具条编写代码是 Windows 应用程序的传统。Delphi 提供了菜单对象和工具条对象, 使用起来比较方便。但是直接在菜单项或工具条按钮的事件中编写代码不是面向对象编程的好习惯, 有的程序员会发现自己经常为菜单项和工具条按钮事件编写同样的代码来完成类似的操作动作, 还有的程序员会发现自己的菜单项或工具条按钮的事件中混杂着大量与界面无关的业务内容, 一旦业务出现变化, 代码改动起来异常困难。实际上 Delphi 有 Action 对象, 用于集中编写实现操作动作的代码, 并在菜单项和工具条按钮的 Action 属性中关联。这样一来, 使用 Action 对象编写的动作代码就完全和界面分开了, 因而其灵活高效, 便于重用。

比如, 在我的图书管理示例程序中, 用户如果需要完成对数据库记录的编辑操作, 可以有 4 种不同的方式打开数据库的编辑窗口:

- 使用主菜单的菜单项: “文件” | “属性”。
- 点击工具条的 “属性” 按钮。
- 右击 ListView 组件的图标, 在弹出菜单中选择菜单项: “属性”。
- 双击 ListView 组件的图标。

这4种方式看似需要编写重复的繁琐的代码,但实际上,我只需将一个打开编辑窗口的 Action 对象(即: actEdit)关联到属性菜单项、工具条的属性按钮、弹出菜单的属性菜单项的 Action 属性中,惟一要写的代码是在 ListView 的双击事件中调用 Action 对象的执行事件:

```
procedure TMainForm.ListView1DbClick(Sender: TObject);  
begin  
    actEdit.Execute(nil);  
end;
```

这就是说,4种不同的方式打开数据库的编辑窗口的动作实际上都是一个 actEdit 的动作,只是提供用户的界面操作形式不一样而已。用户有了多种界面的选择,但我们却将界面的需求和实际的动作分离了,这使程序易于维护。

可喜的是,从 Delphi 6 开始,我们可以使用 ActionManager 组件来统一管理动作(TAction)、菜单(TActionMainMenuBar)和工具条(TActionToolBar)。我本人从 Delphi 6 开始已经不再使用原来的菜单组件(TMainMenu)和工具条(TToolBar)组件,而是用一个 ActionManager 组件就可以全部搞定。图书管理程序中 ActionManager 组件的使用步骤如图 4-8 所示。

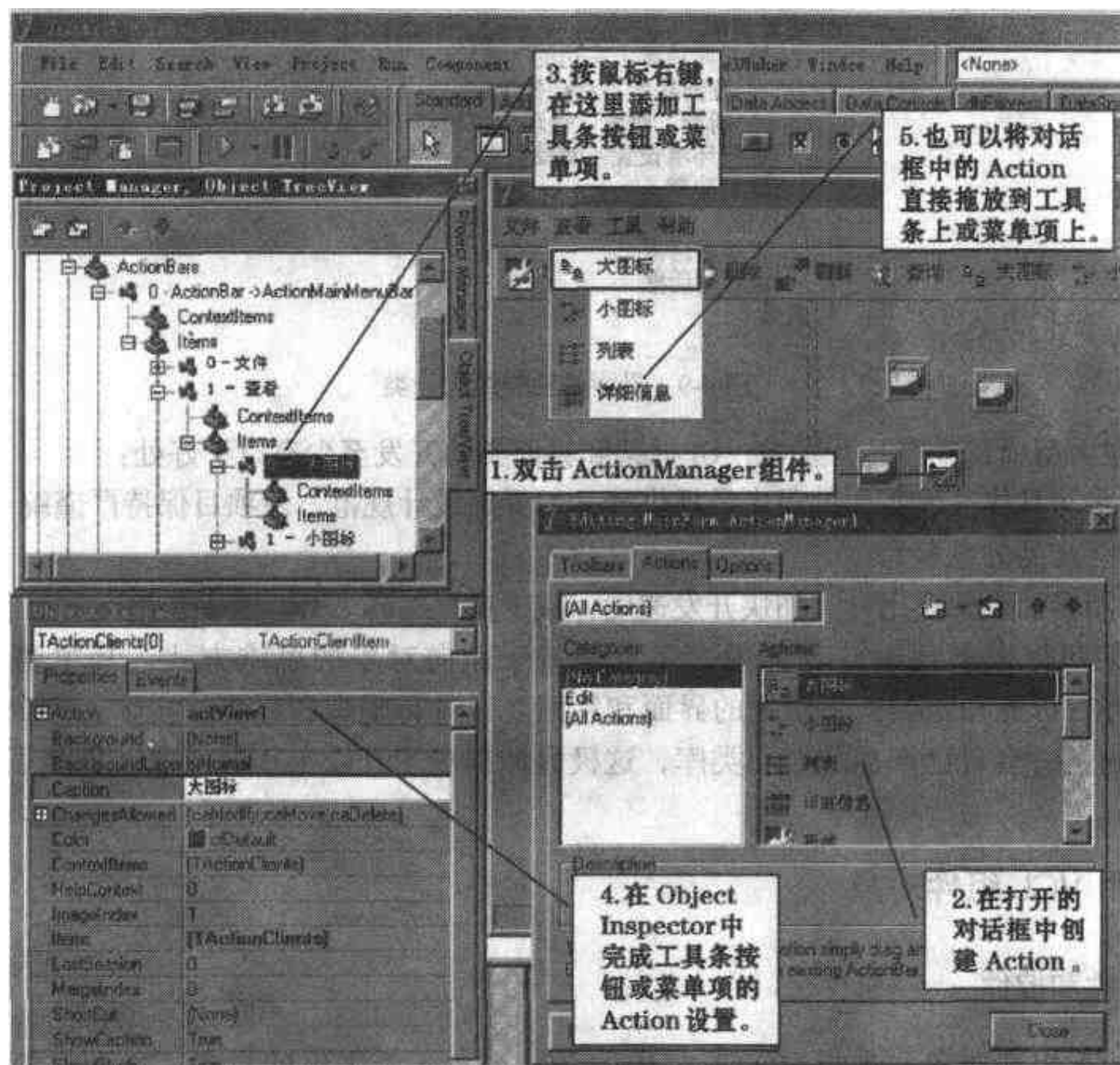


图 4-8 利用 ActionManager 统一管理菜单和工具条

参见 关于 ActionManager 组件的详细用法参见我著的《Delphi 6 企业级解决方案及应用剖析》212 页的“TActionManager”和 529 页的“使用 ActionManager 管理操作动作”。

前面我讨论了界面设计中最重要几个界面要素：窗体布局、功能区划、控件容器、菜单和工具条。实际上这里讨论的只是一个界面的基本组成，界面设计的很多细节还依赖于一些分工不同的各式 UI 组件。虽然 Delphi 的 VCL 提供了大量的 UI 组件，但在商业软件或大型项目中往往不能满足需要。

通常在商业软件或大型项目的开发中需要有一个规范的 UI 框架，这样既可以统一界面和操作，又可以体现重用的原则，满足一些专门的需求。

Windows 应用程序的 UI 框架是基于界面类的。通过分析一些通用和常用的界面操作，我们可以把界面的功能结构做一个分类，如图 4-9 所示。然后根据这个分类再设计出 UI 类库，如图 4-10 所示。

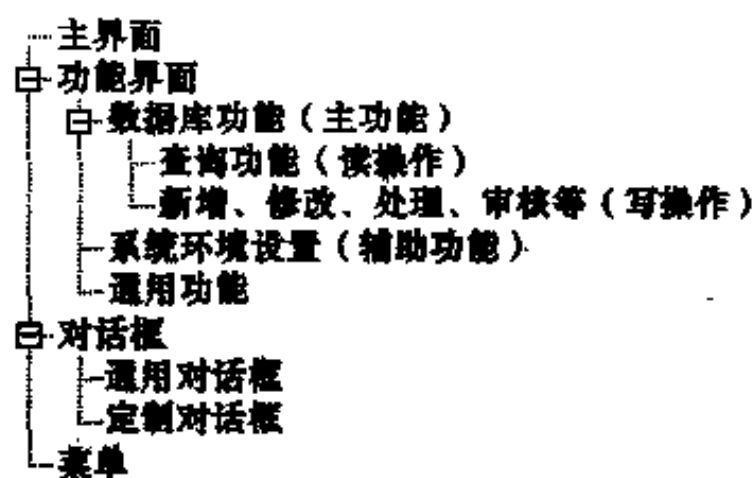


图 4-9 界面的功能结构分类

特别在大型项目或产品开发中，UI 框架和界面类的开发至少有以下好处：

- 为参与具体开发工作的程序员提供统一的界面设计规范，使项目保持严谨统一的风格，增强产品的竞争力。
- 利用界面类的重用性，加快开发速度，减少重复编码。

UI 框架和界面类的设计与开发，完全是基于面向对象的思维方式。其主要工作是对 VCL 组件进行扩充和定制；将常用的界面和界面要素进行整理和封装。实际上整个开发团队最终得到的是一套自己产品的通用类库，这极大地提高了开发工作的效率，增强了产品的可维护性。

4.2 使用 VCL 组件对象

4.2.1 组件和控件

所谓 VCL 组件对象，是指由 VCL 类中的 TComponent 派生出的对象。这种对象大都位于 Delphi 的组件面板上，可以通过拖放和属性设置，进行可视化设计和编程。一个更为有用的方面是它们作为 VCL 组件体现了一种可重用的软件思想，在设计程序时你只需将其看成是具有

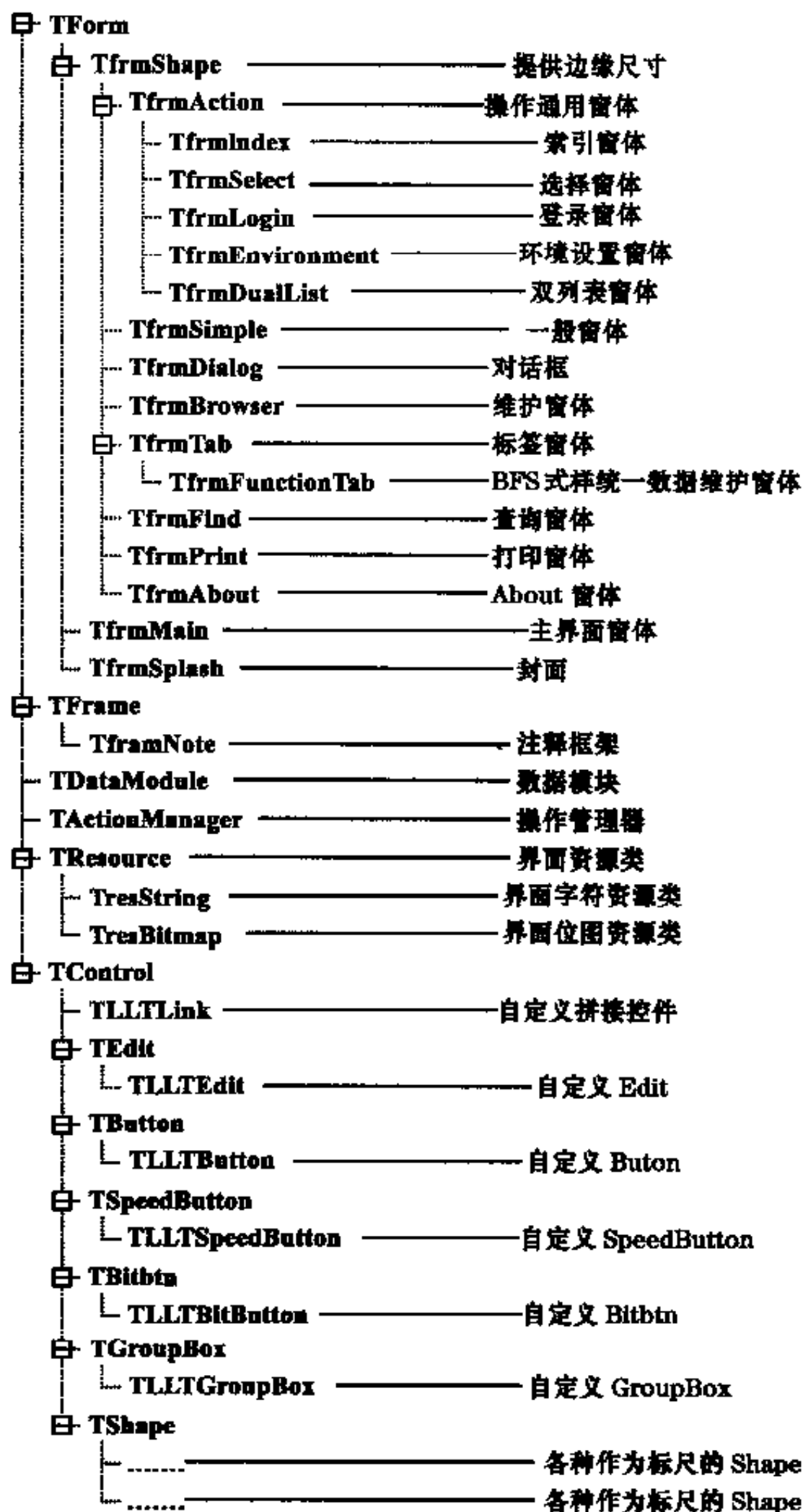


图 4-10 一个由我指导设计的 UI 类库

完成某些功能的“黑箱”。这是一个非常重要的概念，因为从词义上解读 VCL（Visual Component Library）就是“可视化组件库”的意思。不少人可能会将组件（Component）混同于控件（Control），但实际上控件是继承自 TComponent 的派生类 TControl，控件提供了可视化的用户接口，是属于 Windows 标准的一些组件。显然，控件是组件的子集，组件包含控件。

组件之所以在 Delphi 编程中受到欢迎,是因为组件可以在设计时进行可视化修改,而普通的类则需要通过手工编码来修改。组件具有设计精巧的公开接口,这些接口使得大多数组件都具备属性和事件的特性,可以在设计时进行可视化修改,而且对组件属性的改变可以立即生效。这一切归功于 TPersistent 将持久状态的概念引入到类中,即:在应用程序的多次运行之间,可以保存并再次获取类的特性。

因此要记住一点,组件可以是任何一种软件,只要它符合于组件的框架结构——可视化组件库 VCL。无论哪种组件,都必须是从 VCL 类中的某一类生成的。VCL 是完全面向对象的,VCL 中的所有组件和对象都存在着继承和被继承的关系,这种面向对象的方法有许多好处,例如,所有组件能够派生产生新组件,这就使得程序开发者能够根据别人的第三方控件设计出更为专业的第三方控件,而你甚至都不用去管其父类控件的源代码是什么。

注意 使用第三方控件可以提高编程效率,但使用的第三方控件如果存在 Bug,也可能导致应用程序中无法预知的异常。这一点往往不易察觉。所以我建议:第一,千万不要使用来路不明或不熟悉的第三方控件;第二,对于正式项目的开发,选用的第三方控件必须是商业控件或有源代码的控件;第三,使用第三方控件还必须评估其对 Delphi 特定版本依赖性而造成的风险。

对第三方控件有兴趣的读者可以进一步参考本人拙作:

《Delphi 第三方控件使用大全》和《Delphi 第三方控件使用大全 II》(中国水利水电出版社出版)。

4.2.2 组件对象使用实例

目前市面上介绍 VCL 组件使用的书很多,所以我不准备在本书中介绍控件的时尚用法。我只想结合一个具体的应用程序,演示一下组件作为一个对象在程序中是如何使用的。这里还涉及到复合组件、数据绑定、数据同步等问题。

在此采用的示例程序是前面我们提到的图书管理程序,该程序是一个简单的数据库应用程序,但“麻雀虽小,五脏俱全”,它还是能够反映出编程中的许多必须面对的问题。

图书管理程序的主界面是用来浏览图书信息的。如果用一个 DBGrid 组件倒是十分省事,但在组织和导航信息方面功能太弱。更深入的考虑是,直接使用数据库组件来显示数据有一定的局限性,如果数据不是以数据集 (DataSet) 的形式出现,而是以数组、对象集、XML 等形式出现,那么数据库组件就无能为力。在实际应用中,在客户端也不提倡直接通过数据库组件连接到远程数据库上直接显示数据,所以不能过多依赖数据库组件。

我个人比较欣赏微软的 Windows 资源管理器风格的数据导航和浏览风格。所以,我在示例程序中使用 TTreeView 组件实现数据组织和导航,使用 TListView 组件实现数据浏览和操作。不少程序员也喜欢这种方式,不过这两个组件有点复杂,使得他们往往面对一大堆组件的方法和属性,觉得难以下手。

如果我们从面向对象的角度来理解这两个组件可能会比较容易下手。

TTreeView 有一个 TTreeNode 类型的 Items 属性,管理着 TTreeNode 类型的 Item 数组。这就

是说 TTreeNode 组件实际上是 TTreeNode 组件的一个对象集。因此, 为 TTreeView 对象增加节点就是增加 TTreeNode 对象。TTreeNode 对象保持有相关的层次信息, 比如: 父节点对象的信息 (Parent 属性)、子节点对象的信息 (GetXXXChild 方法) 以及相邻节点对象的信息 (GetNext 和 GetPrev 方法)。所以, 使用 TTreeView, 关键是把握好节点对象 TTreeNode 的使用。

示例程序中创建树节点的方法如下。

声明节点对象变量:

```
var  
    RootNode: TTreeNode;
```

创建根节点:

```
tree.Items.Clear;  
RootNode := Tree.Items.Add (nil, '全部'); |Add a root node |  
RootNode.ImageIndex := 1;
```

创建子节点并将数据库查询的结果绑定到子节点上:

```
strQry := 'select distinct 出版社 as a from mybook';  
AddChildNodes ('出版社', strQry, 3);  
strQry := 'select distinct 印次 as a from mybook';  
AddChildNodes ('印次', strQry, 4);  
strQry := 'select distinct 类别 as a from mybook';  
AddChildNodes ('类别', strQry, 5);  
strQry := 'select distinct 出版时间 as a from mybook';  
AddChildNodes ('出版时间', strQry, 6);
```

其中 AddChildNodes 方法如下所示:

```
procedure TMainForm.AddChildNodes (nodeName: String;  
    SQLStr: string; imlID: integer);  
var  
    curID, SubNodeName: string;  
    subnode, subnodechild: TTreeNode;  
begin  
    subnode := tree.Items.AddChild (tree.TopItem, nodeName);  
    subnode.ImageIndex := imlID;  
    with FdmMain.adqBook do  
    begin  
        close;  
        sql.Clear;  
        sql.Add (SQLStr);  
        open;  
        first;  
        while not Eof do  
        begin  
            curID := trim (FieldByName ('a').AsString);  
            SubNodeName := curID;  
            subnodechild := tree.items.addchild (subnode, SubNodeName);  
            subnodechild.ImageIndex := 2;  
            next;  
        end;  
    end;  
end;
```

同样, TListView 也有一个 TListItems 类型的 Items 属性, 管理着 TListItem 类型的 Item 数组。这就是说 TListItems 组件实际上是 TListItem 组件的一个对象集。因此, 为 TListView 对象增加明细项就是增加 TListItem 对象。不同的是 TListItem 对象还有自己的 TStrings 类型的 SubItems 属性, 管理着一个字符串数组作为 TListItem 对象的详细信息。所以, 使用 TListView 关键也就是把握好明细项对象 TListItem 的使用。

示例程序中创建 TListView 明细项并绑定数据库的方法如下:

```
with ListView1 do
begin
  SmallImages := imlSmall;
  LargeImages := imlLarge;
  Clear;
  with FdmMain.adqBook do
  begin
    first;
    for I := 0 to RecordCount - 1 do
    begin
      ListItem := Items.Add;
      ListItem.Caption := FieldByName('图书名称').AsString;
      ListItem.ImageIndex := 2;
      ListItem.SubItems.Add(FieldByName('译著者').AsString);
      ListItem.SubItems.Add(FieldByName('出版社').AsString);
      ListItem.SubItems.Add(FieldByName('出版时间').AsString);
    end;
  end;
end;
```

TListView 之所以作为浏览信息的首选组件, 是因为它提供了 4 种不同的明细项查看方式: 大图标、小图标、列表和详细信息。这是由 TListView 的 ViewStyle 属性决定的。为了调用该功能, 我们为显示大图标、小图标、列表和详细信息这 4 个 TAction 对象 (actView1、actView2、actView3、actView4) 编写了一个共同的事件代码:

```
procedure TMainForm.actViewExecute(Sender: TObject);
begin
  ListView1.ViewStyle := TViewStyle((Sender as TComponent).Tag);
end;
```

其中 TViewStyle 类型定义为: TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport)。为什么 4 个不同的 TAction 对象可以共用一个事件代码? 秘密在于 TAction 对象的 Tag 属性设置, 如图 4-11 所示。该 Tag 属性的设置刚好对应了 TViewStyle 的序数值, 可以转型为 ListView1.ViewStyle 的值。活用 Tag 属性可以带来编程上的便捷, 千万别忽略这个有用的属性。

由此可见, TTreeView 和 TListView 都是一类复合组件, 它们的数据绑定都是基于对象集属性 Items 的。无论元素对象是节点 (TTreeNode 对象) 还是明细项 (TListItem 对象), 都是 Items 的 Item 对象。

除了 TTreeView 和 TListView 组件对象的使用外, 从示例程序其他的编程技巧中你也能看到对象使用的痕迹。比如为了在 TListView 中显示漂亮的背景图案, 我就使用了一个图形对象

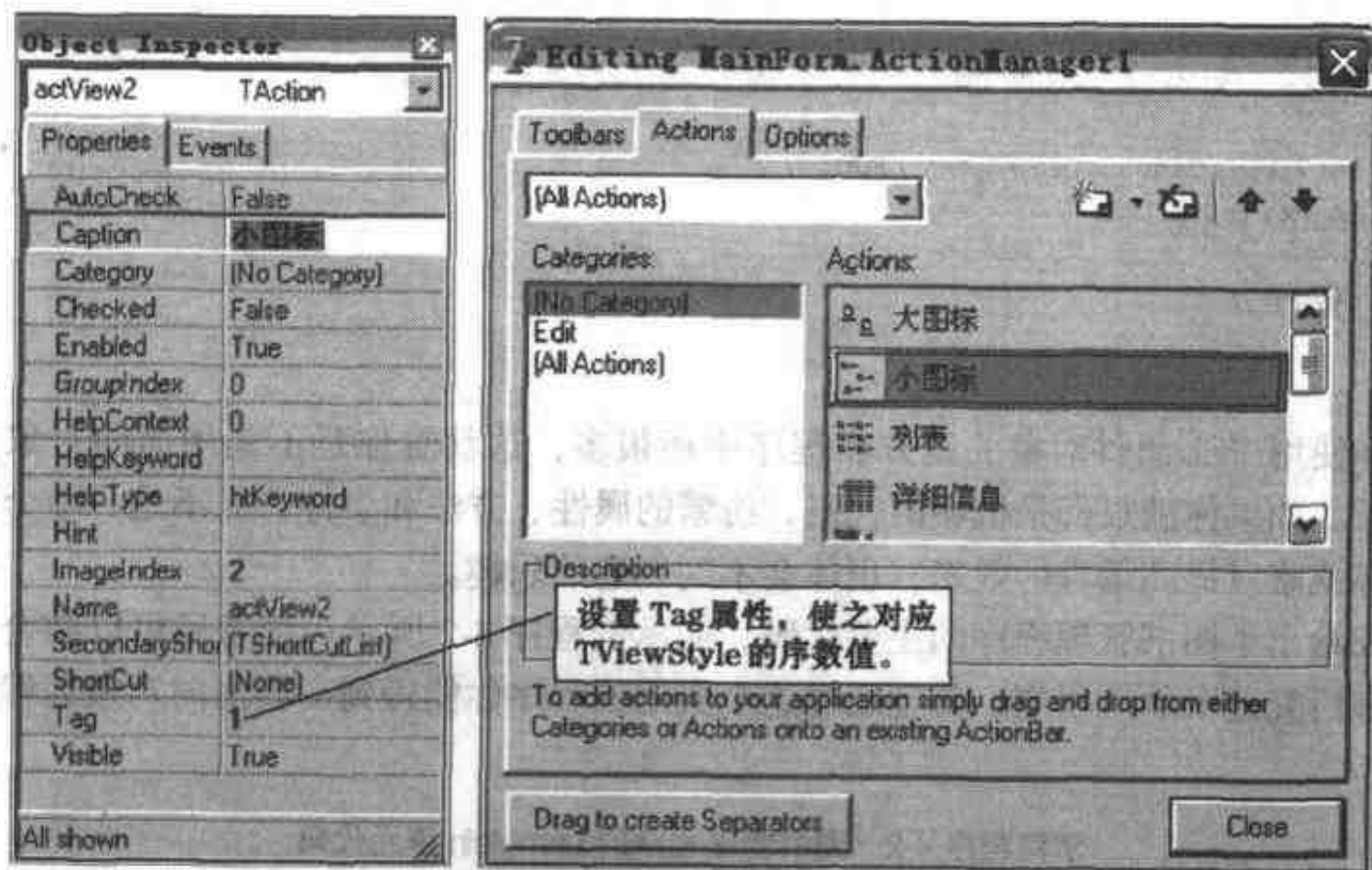


图 4-11 通过设置 Tag 属性，使得 4 个 TAction 对象可以共用一个事件代码

FJpg，它是作为 TMainForm 的数据成员定义在私有域中的：

```
FJpg: TJPEGImage;
```

我在主窗体的 OnCreate 事件中创建了这个图形对象，一同创建的还有数据模块对象。

```
procedure TMainForm.FormCreate (Sender: TObject);
var
  NewColumn: TListColumn;
begin
  FdmMain := TdmMain.Create (self);
  .....
  Fjpg := TJPEGImage.Create;
  Fjpg.LoadFromFile ('bg003.jpg');
end;
```

在 ListView1 的 OnCustomDraw 事件中，一个在 ListView1 的 Canvas 上绘图的小技巧实现了填充背景功能。

```
procedure TMainForm.ListView1CustomDraw (Sender: TCustomListView;
  const ARect: TRect; var DefaultDraw: Boolean);
var
  x, y, w, h: Integer;
begin
  with FJpg do
  begin
    w := Width;
    h := Height;
  end;
  y := 0;
  while y < ListView1.Height do
  begin
```

```

x:= 0;
while x< ListView1.Width do
begin
  ListView1.Canvas.Draw (x,y,FJpg);
  Inc (x,w);
end;
Inc (y,h);
end;
end;

```

像这样使用 VCL 组件对象的地方在程序中还很多，这就看你是不是用面向对象的眼光来阅读程序了。如果你满眼都是机械的代码，纷繁的属性、方法和事件；而不是一个个鲜活有机的对象，那就难以提高编程的效率，也体会不到创造的乐趣。

最后我给出了图书管理程序的主要代码，如示例程序 4-2 所示。读者还可以结合光盘中的整个项目来研究这个示例程序，希望通过这个实用的程序你能得到一些面向对象编程和使用组件的体会。

示例程序 4-2 图书管理程序的 ufrmMain 单元代码

```

unit ufrmMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ActnMenus, ToolWin, ActnMan, ActnCtrls, StdActns, ActnList,
  XPStyleActnCtrls, ImgList, ExtCtrls, ComCtrls, jpeg, DB, DBCtrls, Menus, uDM;

type
  TMainForm = class (TForm)
    ToolbarImages: TImageList;
    ActionManager1: TActionManager;
    EditCopy1: TEditCopy;
    EditPaste1: TEditPaste;
    EditCut1: TEditCut;
    EditSelectAll1: TEditSelectAll;
    actView1: TAction;
    actView2: TAction;
    actView3: TAction;
    actView4: TAction;
    actNew: TAction;
    actEdit: TAction;
    actDelete: TAction;
    actFind: TAction;
    actPrint: TAction;
    actExit: TAction;
    ActionMainMenuBar1: TActionMainMenuBar;
    StatusBar1: TStatusBar;
    ActionToolBar2: TActionToolBar;
    Panel1: TPanel;
    Splitter1: TSplitter;
    Panel2: TPanel;
  end;

```

```

Tree: TTreeView;
ListView1: TListView;
imlSmall: TImageList;
imlLarge: TImageList;
Splitter2: TSplitter;
Image1: TImage;
actRefresh: TAction;
PopupMenu1: TPopupMenu;
N1: TMenuItem;
N2: TMenuItem;
N3: TMenuItem;
N4: TMenuItem;
procedure FormCreate (Sender: TObject);
procedure TreeClick (Sender: TObject);
procedure actViewExecute (Sender: TObject);
procedure ListView1CustomDraw (Sender: TCustomListView;
    const ARect: TRect; var DefaultDraw: Boolean);
procedure FormDestroy (Sender: TObject);
procedure ListView1SelectItem (Sender: TObject; Item: TListItem;
    Selected: Boolean);
procedure actEditExecute (Sender: TObject);
procedure actExitExecute (Sender: TObject);
procedure actNewExecute (Sender: TObject);
procedure actRefreshExecute (Sender: TObject);
procedure actDeleteExecute (Sender: TObject);
procedure ListView1DbClick (Sender: TObject);
procedure FormResize (Sender: TObject);
private
    FJpg: TJPEGImage;
    procedure AddChildNodes (nodeName: String; SQLStr: string; imlID: integer);
public
    FdmMain: TdmMain;
end;

var
    MainForm: TMainForm;

implementation

uses ufrmEdit;

{$R *.dfm}

procedure TMainForm.AddChildNodes (nodeName: String;
    SQLStr: string; imlID: integer);
var
    curID, SubNodeName: string;
    subnode, subnodechild: TTreeNode;
begin
    subnode: = tree.Items.AddChild (tree.TopItem, nodeName);
    subnode.ImageIndex: = imlID;
    with FdmMain.adqBook do
    begin

```

```

    close;
    sql.Clear;
    sql.Add (SQLStr);
    open;
    first;
    while not Eof do
    begin
        curID: = trim (FieldByName ('a') .AsString);
        SubNodeName: = curID;
        subnodechild: = tree.items.addchild (subnode, SubNodeName);
        subnodechild.ImageIndex: = 2;
        next;
    end;
end;
end;
end;

```

```

procedure TMainForm.FormCreate (Sender: TObject);

```

```

var
    NewColumn: TListColumn;
begin
    FdmMain: = TDMMain.Create (self);
    actRefreshExecute (nil);
    with ListView1 do
    begin
        SmallImages: = imlSmall;
        LargeImages: = imlLarge;
        NewColumn: = Columns.Add;
        NewColumn.Caption: = '书名';
        NewColumn.Width: = 280;
        NewColumn: = Columns.Add;
        NewColumn.Caption: = '译著者';
        NewColumn.Width: = 80;
        NewColumn: = Columns.Add;
        NewColumn.Caption: = '出版社';
        NewColumn.Width: = 100;
        NewColumn: = Columns.Add;
        NewColumn.Caption: = '出版时间';
        NewColumn.Width: = 80;
    end;
    Fjpg: = TJPEGImage.Create;
    Fjpg.LoadFromFile ('bg003.jpg');
end;

```

```

procedure TMainForm.TreeClick (Sender: TObject);

```

```

var
    a, b: string;
    I: Integer;
    ListItem: TListItem;
begin
    a: = tree.Selected.Text;
    with FdmMain.adqBook do
    begin
        Filtered: = false;
    end;
end;

```



```
if (a < > '全部') then
begin
  b := tree.Selected.Parent.Text;
  Filtered := false;
  if (b < > '全部') then
  begin
    Filter := b + '=' + #39 + a + #39;
    Filtered := True;
  end;
end;
end;

with ListView1 do
begin
  SmallImages := imlSmall;
  LargeImages := imlLarge;
  Clear;
  with FdmMain.adqBook do
  begin
    first;
    for I := 0 to RecordCount - 1 do
    begin
      ListItem := Items.Add;
      ListItem.Caption := FieldByName('图书名称').AsString;
      ListItem.ImageIndex := 2;
      ListItem.SubItems.Add(FieldByName('译著者').AsString);
      ListItem.SubItems.Add(FieldByName('出版社').AsString);
      ListItem.SubItems.Add(FieldByName('出版时间').AsString);
    end;
  end;
end;

procedure TMainForm.actViewExecute(Sender: TObject);
begin
  ListView1.ViewStyle := TViewStyle((Sender as TComponent).Tag);
end;

procedure TMainForm.ListView1CustomDraw(Sender: TCustomListView;
  const ARect: TRect; var DefaultDraw: Boolean);
var
  x, y, w, h: Integer;
begin
  with FJpg do
  begin
    w := Width;
    h := Height;
  end;
  y := 0;
  while y < ListView1.Height do
  begin
    x := 0;
    while x < ListView1.Width do
```

```

begin
  ListView1.Canvas.Draw (x, y, FJpg);
  Inc (x, w);
end;
Inc (y, h);
end;
end;

procedure TMainForm.FormDestroy (Sender: TObject);
begin
  FJpg.free;
end;

procedure TMainForm.ListView1SelectItem (Sender: TObject; Item: TListItem;
  Selected: Boolean);
begin
  FdmMain.adqBook.Locate ('图书名称', Item.Caption, [loPartialKey]);
  TBlobField (FdmMain.adqBook.FieldByName ('封面照片')) .SaveToFile ('1.bmp');
  Image1.Picture.LoadFromFile ('1.bmp');
end;

procedure TMainForm.actEditExecute (Sender: TObject);
var
  EditForm: TEditForm;
  NeedRefresh: Boolean;
begin
  NeedRefresh: = False;
  EditForm: = TEditForm.Create (nil);
  try
    FdmMain.adqBook.edit;
    if (EditForm.ShowModal = mrOK) then
      begin
        FdmMain.adqBook.post;
        NeedRefresh: = True;
      end
    else
      FdmMain.adqBook.Cancel;
  finally
    EditForm.Free;
  end;
  if NeedRefresh then
    begin
      FdmMain.adqBook.Filtered: = false;
      Tree.FullCollapse;
    end;
  end;

procedure TMainForm.actExitExecute (Sender: TObject);
begin
  close;
end;

procedure TMainForm.actNewExecute (Sender: TObject);
var

```

```

    EditForm: TEditForm;
    NeedRefresh: Boolean;
begin
    NeedRefresh := True;
    EditForm := TEditForm.Create (nil);
    try
        FdmMain.adqBook.Append;
        if (EditForm.ShowModal = mrOK) then
            begin
                try
                    FdmMain.adqBook.Post;
                except
                    on e: exception do
                        begin
                            if e.Message = 'Empty row cannot be inserted.
                                Row must have at least one column value set' then
                                application.MessageBox ('没有输入内容的空记录无效!',
                                    '提示', MB_ICONWARNING);
                            FdmMain.adqBook.Cancel;
                            NeedRefresh := False;
                        end;
                    end;
                end
            end
        else
            begin
                FdmMain.adqBook.Cancel;
                NeedRefresh := False;
            end;
        finally
            EditForm.Free;
        end;
        if NeedRefresh then
            actRefreshExecute (nil);
    end;

procedure TMainForm.actRefreshExecute (Sender: TObject);
var
    strQry: string;
    RootNode: TTreeNode;
begin
    FdmMain.adqBook.Filtered := false;
    tree.Items.Clear;
    RootNode := Tree.Items.Add (nil, '全部'); {Add a root node}
    RootNode.ImageIndex := 1;
    strQry := 'select distinct 出版社 as a from mybook';
    AddChildNodes ('出版社', strQry, 3);
    strQry := 'select distinct 印次 as a from mybook';
    AddChildNodes ('印次', strQry, 4);
    strQry := 'select distinct 类别 as a from mybook';
    AddChildNodes ('类别', strQry, 5);
    strQry := 'select distinct 出版时间 as a from mybook';
    AddChildNodes ('出版时间', strQry, 6);
    with FdmMain.adqBook do
        begin

```

```
close;
sql.Clear;
sql.Add ('select * from mybook order by 图书名称');
open;
StatusBar1.Panels [1].Text: = '共有记录' + inttostr (RecordCount) + '条。';
end;
StatusBar1.Panels [0].Text: = DateTimeToStr (now);
end;

procedure TMainForm.actDeleteExecute (Sender: TObject);
var
  str: Pchar;
begin
  str: = Pchar ('是否删除记录 ('
    + FdmMain.adgBook.FieldName ('图书名称') .AsString + ')? ');
  if (Application.MessageBox (str, '提示', MB_YESNO + MB_ICONQUESTION) = IDYES)
  then
    begin
      FdmMain.adgBook.Delete;
      FdmMain.adgBook.Post;
    end;
end;

procedure TMainForm.ListView1DblClick (Sender: TObject);
begin
  actEditExecute (nil);
end;

procedure TMainForm.FormResize (Sender: TObject);
begin
  imagel.Update;
end;

end.
```

4.2.3 组件使用的误区

示例程序 4-2 是主界面的全部源码, 看似复杂的 Windows 资源管理器风格的主界面, 其实并没有太多的实现代码。除了编程上的精炼, 还要感谢 Delphi 这一功能强大的开发工具。因为很多功能 VCL 组件已经帮你实现, 却不要你写一行代码!

但是在 Delphi 编程中有一种现象实在令人担忧, 那就是滥用组件, 使用一些画蛇添足式的技巧。我在一本《Delphi 技巧精选实例集》的书上看到一个称之为“在 DBGrid 中插入下拉组合框”的技巧。所谓在 DBGrid 中插入下拉组合框, 实际上是将一个 DBComboBox1 动态移动到 DBGrid 的当前输入网格上, 并通过 DBComboBox1 的下拉组合框来选择输入项。说白了就是“障眼法”。

图 4-12 是“在 DBGrid 中插入下拉组合框”技巧的设计界面, 示例程序 4-3 是源代码。虽然这种设计能满足一定的需要, 但却完全没有必要, 因为 DBGrid 自身就可以实现这一功能。

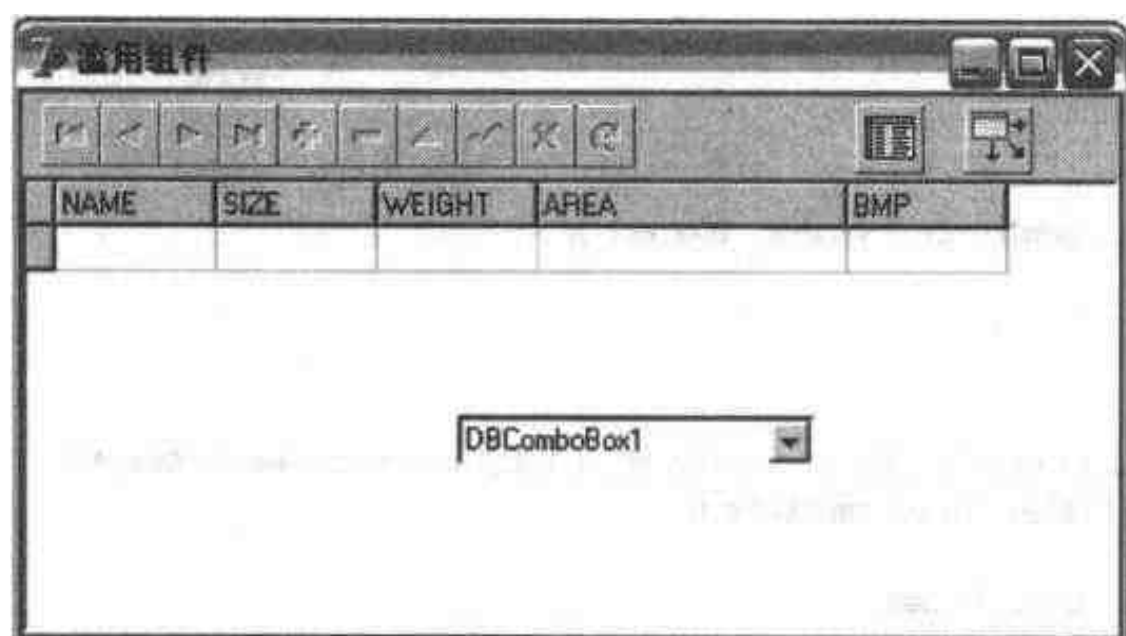


图 4-12 “在 DBGrid 中插入下拉组合框”的设计界面

示例程序 4-3 “在 DBGrid 中插入下拉组合框”的源程序

```

unit Unit1;

interface

uses
  Windows, Messages, Classes, SysUtils, Graphics, Controls, StdCtrls, Forms,
  Dialogs, DBCtrls, DB, DBGrids, DBTables, Grids, ExtCtrls;

type
  TForm1 = class (TForm)
    Table1NAME: TStringField;
    Table1SIZE: TSmallintField;
    Table1WEIGHT: TSmallintField;
    Table1AREA: TStringField;
    Table1BMP: TBlobField;
    DBGrid1: TDBGrid;
    DBNavigator: TDBNavigator;
    Panel1: TPanel;
    DataSource1: TDataSource;
    Panel2: TPanel;
    Table1: TTable;
    DBComboBox1: TDBComboBox;
    procedure FormCreate (Sender: TObject);
    procedure DBGrid1DrawDataCell (Sender: TObject; const Rect: TRect;
      Field: TField; State: TGridDrawState);
    procedure DBGrid1ColExit (Sender: TObject);
    procedure DBGrid1KeyPress (Sender: TObject; var Key: Char);
  private
    {private declarations}
  public
    {public declarations}
  end;

var
  Form1: TForm1;

```

```

implementation

{$R *.DFM}

procedure TForm1.FormCreate (Sender: TObject);
begin
    Table1.Open;
end;

procedure TForm1.DBGrid1DrawDataCell (Sender: TObject; const Rect: TRect;
    Field: TField; State: TGridDrawState);
begin
    if (gdFocused in State) then
        if (Field.FieldName = DBComboBox1.DataField) then
            begin
                DBComboBox1.Left := Rect.Left + DBGrid1.Left;
                DBComboBox1.Top := Rect.Top + DBGrid1.Top;
                DBComboBox1.Width := Rect.Right - Rect.Left;
                DBComboBox1.Height := Rect.Bottom - Rect.Top;
                DBComboBox1.Visible := True;
            end;
        end;
end;

procedure TForm1.DBGrid1ColExit (Sender: TObject);
begin
    If DBGrid1.SelectedField.FieldName = DBComboBox1.DataField then
        begin
            DBComboBox1.Visible := false;
        end;
end;

procedure TForm1.DBGrid1KeyPress (Sender: TObject; var Key: Char);
begin
    if (key < > chr (9)) then
        if (DBGrid1.SelectedField.FieldName = DBComboBox1.DataField) then
            begin
                DBComboBox1.SetFocus;
                SendMessage (DBComboBox1.Handle, WM_Char, word (Key), 0);
            end;
        end;
end;

end.

```

对于 TDBGrid 组件，研究它的 TColumn 对象就会发现它有一个 TStrings 类型的 PickList 属性。该属性的定义和实现代码分别如下：

```

property PickList: TStrings read GetPickList write SetPickList;

function TColumn.GetPickList: TStrings;
begin
    if FPickList = nil then
        FPickList := TStringList.Create;
    Result := FPickList;
end;

```

```
procedure TColumn.SetPickList (Value: TStrings);
begin
  if Value = nil then
  begin
    FPickList.Free;
    FPickList := nil;
    Exit;
  end;
  PickList.Assign (Value);
end;
```

显然，PickList 属性封装了一个 TStrings 类型的对象，通过设置 PickList 同样可以实现 DBGrid 的下拉组合框输入，如图 4-13 所示。



图 4-13 通过设置 PickList 同样可以实现 DBGrid 的下拉组合框输入

实际上，PickList 才是真正插入在 DBGrid 中的下拉组合框。虽然 PickList 属性可以在设计期进行可视化设置，但实际编程中我们可以将一个 TStrings 类型的对象动态赋值给 PickList 属性，发挥它的强大功能。

所以，不去好好了解 VCL 组件的内在功能，而去追求所谓的技巧，是组件使用的一大误区。显然，滥用组件和技巧不但浪费了资源，还无谓地增加了代码的复杂性和编程的工作量。

4.3 使用对象集

在程序执行当中，有一些对象是我们在设计时就已经确定的，还有一些是运行时动态产生的。无论是哪一种情况产生的对象，我们都需要对它们进行管理。这就意味着，我们必须有能力在任何时间、任何地方创建它们，并在不需要它们时进行清理。

对象的管理实际上是对对象引用的管理。使用对象数组以及有内部对象数组机制的容器对象，可以完成针对对象集的操作和管理。

4.3.1 对象数组

通过前面介绍的 TDBGrid、TTreeView 和 TListView 组件，我们可以发现它们都是一类复合组

件，它们的数据绑定都是基于对象集属性 `Items` 的。所谓 `Items` 属性反映了 Delphi 中对象数组的应用，即 `Items` 管理着一个内部的对象数组，该数组包含了 `Item` 对象。这种对象数组的对应关系随处可以找到，比如：`TTreeView` 的节点（`TTreeNode`——`TTreeNode`）、`TListView` 的明细项（`TListItem`——`TListItem` 对象），`TDBGrid` 的列（`TColumn`——`TColumn`）等，都是 `Items`——`Item` 关系的变形。因此，我们可以得出这样的结论，对象集（或对象数组）在编程中的用处非常广，VCL 组件就是我们学习的最好范例。

由于 Delphi 中有各种现成的具备对象数组功能的类可以使用，因此我们完全没有必要自行创建一个对象数组。读者不妨来查看一下 Delphi 中一个简单实用的对象数组类 `TObjectList`。`TObjectList` 来自 Delphi 运行时库（Delphi Runtime Library）的 `Contnrs` 单元（需要 `Uses` 该单元），它继承自 `TList`，又是 `TComponentList` 的基类，如图 4-14 所示。

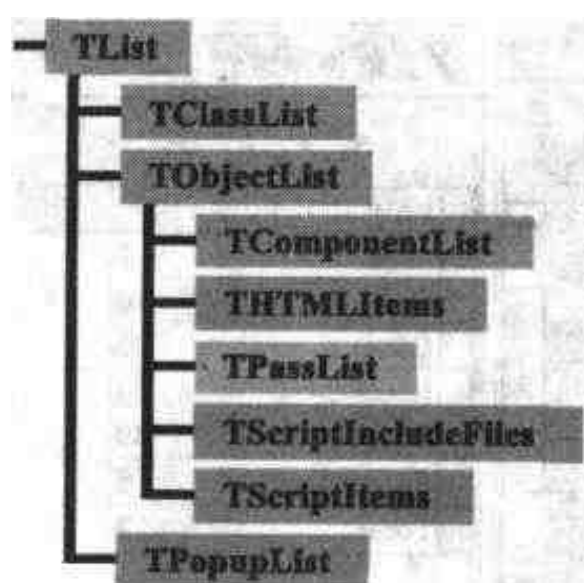


图 4-14 `TList` 及其派生类

`TObjectList` 的声明如示例程序 4-4 所示。

示例程序 4-4 `TObjectList` 的声明

```

TObjectList = class (TList)
private
  FOwnsObjects: Boolean;
protected
  procedure Notify (Ptr: Pointer; Action: TListNotification); override;
  function GetItem (Index: Integer): TObject;
  procedure SetItem (Index: Integer; AObject: TObject);
public
  constructor Create; overload;
  constructor Create (AOwnsObjects: Boolean); overload;
  function Add (AObject: TObject): Integer;
  function Extract (Item: TObject): TObject;
  function Remove (AObject: TObject): Integer;
  function IndexOf (AObject: TObject): Integer;
  function FindInstanceOf (AClass: TClass; AExact: Boolean = True;
    AStartAt: Integer = 0): Integer;
  procedure Insert (Index: Integer; AObject: TObject);
  function First: TObject;
  function Last: TObject;
  property OwnsObjects: Boolean read FOwnsObjects write FOwnsObjects;

```

```

property Items[Index: Integer]: TObject read GetItem write SetItem;
default;
end;

```

显然，TObjectList 不仅可以作为对象数组使用，而且还提供了管理对象集的许多属性和方法。下面我就来举一个实用的例子，介绍 TObjectList 的使用。

示例程序是一个系统登录的例子。它的主要功能是将数据集转化为对象集。这样一来，ClientDataSet1 的每一条数据记录就变成了 ProfileList 对象数组中的每一个对象。然后再实现用户登录。

将数据集转化为对象集是面向对象数据库设计的重要技术之一。这样原来对记录的操作就变成了对对象的操作。这个例子本是我的一个多层分布式应用系统的一部分，只不过为了方便读者学习进行了简化，将 ClientDataSet1 的远程数据库服务连接改为使用本地的一个文件。示例程序的设计界面如图 4-15 所示，程序代码如示例程序 4-5 所示。

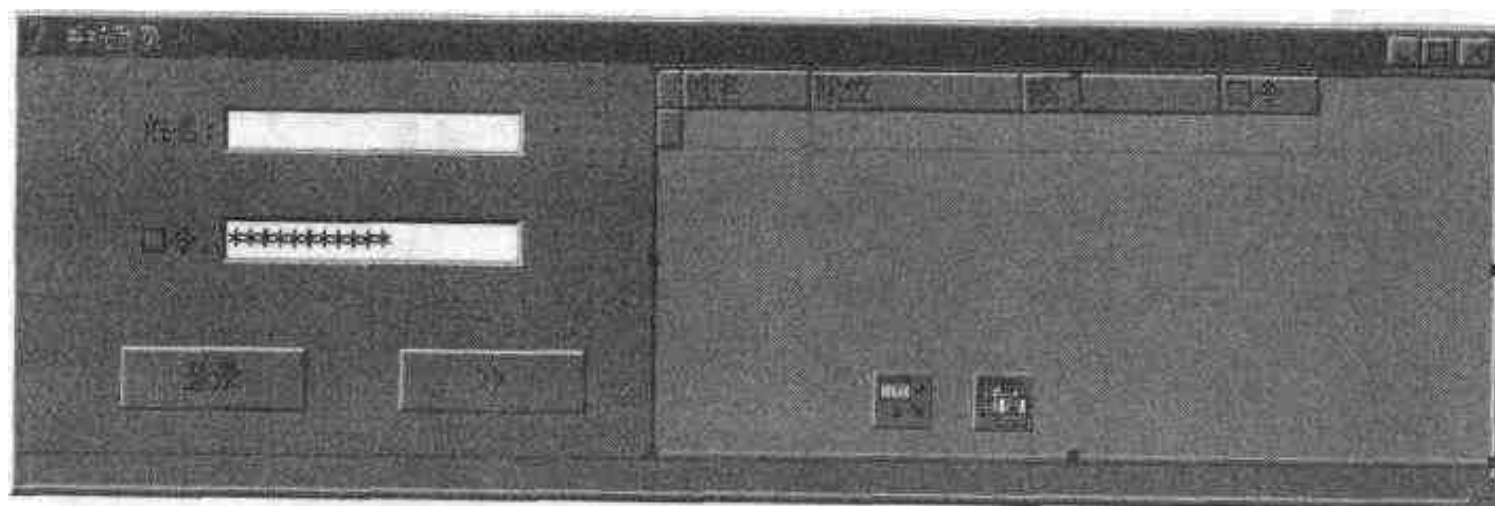


图 4-15 系统登录的设计界面

示例程序 4-5 系统登录程序的源代码

```

unit login;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Mask, ExtCtrls, DB, Grids, DBGrids, Contnrs,
  DBClient, ComCtrls;

type
  TForm1 = class (TForm)
    edtName: TLabeledEdit;
    edtPassword: TMaskEdit;
    Label1: TLabel;
    ClientDataSet1: TClientDataSet;
    ClientDataSet1autoID: TAutoIncField;
    ClientDataSet1ID: TWideStringField;
    ClientDataSet1NAME: TWideStringField;
    ClientDataSet1SEX: TWideStringField;
    ClientDataSet1JOB: TWideStringField;
    ClientDataSet1TEL: TWideStringField;

```

```

ClientDataSet1CALL: TWideStringField;
ClientDataSet1DEP: TWideStringField;
ClientDataSet1GROUP_ID: TWideStringField;
ClientDataSet1PASSWORD: TWideStringField;
DataSource1: TDataSource;
btnLogin: TButton;
DBGGrid1: TDBGGrid;
btnHint: TButton;
StatusBar1: TStatusBar;
procedure FormCreate (Sender: TObject);
procedure btnLoginClick (Sender: TObject);
procedure btnHintClick (Sender: TObject);
private
    ProfileList: TObjectList;
public
    {Public declarations }
end;

TProfile = class (TObject)
    Name : String;
    Dep : String;
    Password : String;
    Job : String;
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate (Sender: TObject);
var
    i: integer;
    AMan: TProfile;
begin
    DBGGrid1.Visible := false;
    self.Width := 270;
    ClientDataSet1.LoadFromFile (' login.dat ');
    ClientDataSet1.Active := True;
    ProfileList := TObjectList.Create (True);
    for i := 1 to ClientDataSet1.RecordCount do
    begin
        AMan := TProfile.Create;
        AMan.Name := ClientDataSet1.FieldName (' Name ') .AsString;
        AMan.Job := ClientDataSet1.FieldName (' Job ') .AsString;
        AMan.Dep := ClientDataSet1.FieldName (' Dep ') .AsString;
        AMan.Password := ClientDataSet1.FieldName (' Password ') .AsString;
        ProfileList.Add (AMan);
        ClientDataSet1.Next;
    end;
end;
end;

```

```
procedure TForm1.btnLoginClick (Sender: TObject);
var
  i: integer;
  AMan: TProfile;
begin
  for i := 0 to (ProfileList.Count-1) do
  begin
    AMan := TProfile (ProfileList.Items [i] );
    if (Trim (AMan.Name) = Trim (edtName.Text)) and
      (Trim (AMan.Password) = Trim (edtPassword.Text)) then
    begin
      application.MessageBox ('登录成功。','提示', MB_OK + MB_ICONINFORMATION);
      StatusBar1.Panels [0] .Text: = '当前用户: ' + AMan.Name + AMan.Dep + AMan.Job;
      exit;
    end;
  end;
  application.MessageBox ('非法用户,登录失败!','提示', MB_OK + MB_ICONSTOP);
end;

procedure TForm1.btnHintClick (Sender: TObject);
begin
  if DBGrid1.Visible then
  begin
    self.Width: = 270;
    btnHint.Caption: = '> >';
  end
  else
  begin
    self.Width: = 620;
    btnHint.Caption: = '< <';
  end;
  DBGrid1.Visible: = not DBGrid1.Visible;
end;

end.
```

TObjectList 管理内部对象数组的功能十分强大。作为数组成员的对象有着自己的生命期, 如果将 TObjectList 的 OwnsObjects 属性设为 True, 可以让 TObjectList 自动管理数组成员对象的生命期, 也可以利用重载的 Create 方法, 在创建 TObjectList 对象时就设置好 OwnsObjects 属性。例如:

```
ProfileList := TObjectList.Create (True);
```

通过 TObjectList 的 Add、Remove、Insert 等方法还可以增加、删除和插入对象。实际上 TObjectList 管理对象数组的这些方法都是继承自基类 TList。

对于一些需要处理字符串数组的组件, 如: TComboBox 和 TListBox, 它们使用的是 TStringList 类型的 Items 属性。TStringList 继承自 TStrings, 覆盖了 TStrings 的加入 (Add)、插入 (Insert)、删除 (Delete) 和查找索引 (indexOf) 等方法。另外覆盖的还有 TStrings 类的两个重要的方法 LoadFromFile 和 SaveToFile。SaveToFile 可将 TStrings 所存的字符串存成文件, 而 LoadFromFile 可将文件中的字符串读入。同样也可将集合中的值写到流或从流中读出 (见图 4-16)。

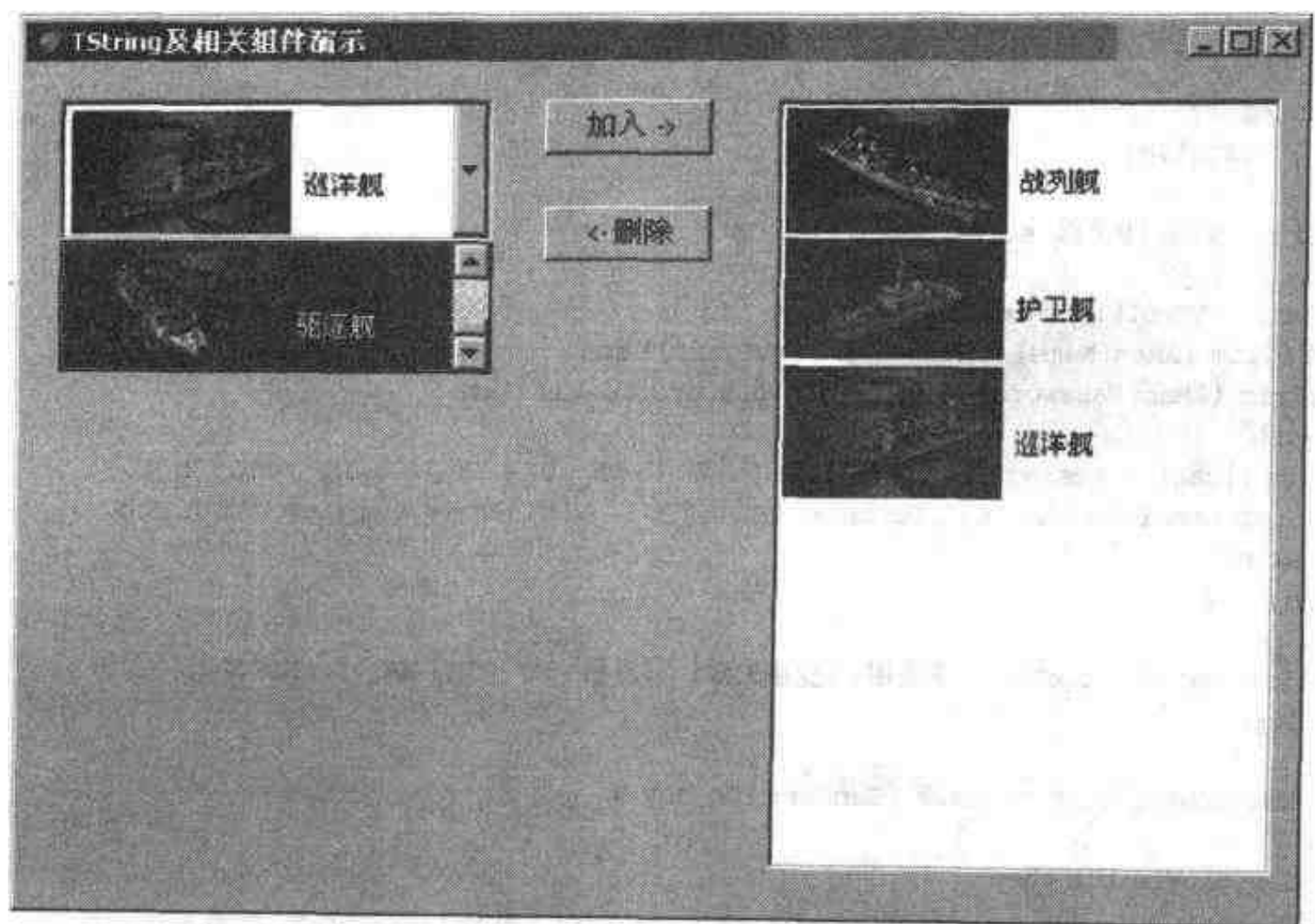


图 4-16 TString 及其相关组件演示程序运行界面

有意思的是 TStringList 类不但能管理内部的字符串数组，还提供了对对象数组的支持。TStringList 类在内部分别为字符串和对象各声明了一个数组来存储数据，因此 TStringList 能同时保存字符串和对象。另外，还可在 Object Inspector 中设置字符串的初始值，使应用程序的代码更简单。示例程序 4-6 以对象集的管理来说明 TStringList 类的使用。

示例程序 4-6 TString 及其相关组件演示程序运行

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class (TForm)
    ComboBox1: TComboBox;
    ListBox1: TListBox;
    btnAdd: TButton;
    btnDelete: TButton;
    procedure FormCreate (Sender: TObject);
    procedure FormDestroy (Sender: TObject);
    procedure ComboBox1DrawItem (Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ComboBox1MeasureItem (Control: TWinControl; Index: Integer;
      var Height: Integer);
```

```

    procedure FormClose (Sender: TObject; var Action: TCloseAction);
    procedure ListBox1DrawItem (Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ListBox1MeasureItem (Control: TWinControl; Index: Integer;
      var Height: Integer);
    procedure btnAddClick (Sender: TObject);
    procedure btnDeleteClick (Sender: TObject);
private
    FItems: TStrings;
public
    Property MyItems: TStrings read FItems write FItems;
end;

var
    Form1: TForm1;
    Pic1, Pic2, Pic3, Pic4, Pic5: TBitmap;
implementation

{$R *.dfm}

procedure TForm1.FormCreate (Sender: TObject);
begin
    Pic1 := TBitmap.Create;
    Pic1.LoadFromFile ('s1.bmp');
    Pic2 := TBitmap.Create;
    Pic2.LoadFromFile ('s2.bmp');
    Pic3 := TBitmap.Create;
    Pic3.LoadFromFile ('s3.bmp');
    Pic4 := TBitmap.Create;
    Pic4.LoadFromFile ('s4.bmp');
    Pic5 := TBitmap.Create;
    Pic5.LoadFromFile ('s5.bmp');
    MyItems := TStringlist.Create;
    with MyItems do
    begin
        AddObject ('航空母舰', Pic1);
        AddObject ('战列舰', Pic2);
        AddObject ('巡洋舰', Pic3);
        AddObject ('护卫舰', Pic4);
        AddObject ('驱逐舰', Pic5);
    end;
    ComboBox1.Items := MyItems;
    ComboBox1.DropDownCount := 5;
    ComboBox1.ItemIndex := 0;
end;

procedure TForm1.FormDestroy (Sender: TObject);
var i: Integer;
begin
    for i := 1 to 5 do
        FindComponent ('Pic' + IntToStr (i)) .Free;
    end;
end;

procedure TForm1.ComboBox1DrawItem (Control: TWinControl; Index: Integer;

```

```

    Rect: TRect; State: TOwnerDrawState);
var
    Pic: TBitmap;
    LeftOffset, TopOffset: Integer;
begin
    with TComboBox (Control) .Canvas do
    begin
        FillRect (Rect);
        Pic: = TBitmap (MyItems.Objects [Index] );
        if Pic < > nil then
        begin
            BrushCopy (Bounds (Rect.Left + 2, Rect.Top + 2, Pic.Width, Pic.Height) ,
                Pic, Bounds (0, 0, Pic.Width, Pic.Height) , clRed);
            LeftOffset: = Pic.Width + 8;
            TopOffset: = Pic.Height div 2;
        end;
        TextOut (Rect.Left + LeftOffset, Rect.Top + TopOffset, MyItems [Index] );
    end;
end;

procedure TForm1.ComboBox1MeasureItem (Control: TWinControl; Index: Integer;
    var Height: Integer);
begin
    Height: = Pic1.Height;
end;

procedure TForm1.FormClose (Sender: TObject; var Action: TCloseAction);
var i: Integer;
begin
    for i: = 1 to 5 do
        TBitmap (FindComponent ('Pic' + IntToStr (i))) .Free;
    end;

procedure TForm1.ListBox1DrawItem (Control: TWinControl; Index: Integer;
    Rect: TRect; State: TOwnerDrawState);
var
    Pic: TBitmap;
    LeftOffset, TopOffset: Integer;
begin
    with TListBox (control) .Canvas do
    begin
        FillRect (Rect);
        Pic: = TBitmap ( (Control as TListBox) .Items.Objects [Index] );
        if Pic < > nil then
        begin
            BrushCopy (Bounds (Rect.Left + 2, Rect.Top + 2, Pic.Width, Pic.Height) ,
                Pic, Bounds (0, 0, Pic.Width, Pic.Height) , clRed);
            LeftOffset: = Pic.Width + 8;
            TopOffset: = Pic.Height div 2;
        end;
        TextOut (Rect.Left + LeftOffset, Rect.Top + TopOffset, (Control as TListBox) .Items [Index] );
    end;
end;
end;

```



```
procedure TForm1.ListBox1MeasureItem (Control: TWinControl; Index: Integer;  
    var Height: Integer);  
begin  
    Height := Pic1.Height;  
end;  
  
procedure TForm1.btnAddClick (Sender: TObject);  
var i: integer;  
begin  
    i := ComboBox1.ItemIndex;  
  
    ListBox1.Items.AddObject (ComboBox1.Items [i], ComboBox1.Items.Objects [i]);  
end;  
  
procedure TForm1.btnDeleteClick (Sender: TObject);  
var i: integer;  
begin  
    i := ListBox1.ItemIndex;  
    ListBox1.Items.Delete (i);  
end;  
  
end.
```

从示例程序 4-6 中可见,在用 AddObject 向 TStringList 加入对象时,同时要加入一与对象相关联的字符串,即对象的名,用此字符串可查找对象。

通过这些常用的数据集对象的使用,我们可以更加深入地了解一些重要的控件用法,提高编程的效率。

参见 要深入了解 VCL 的对象集组件,请参见 10.2 节。

4.3.2 容器对象

容器对象相当于一个放置对象的容器 (container)。在 Delphi 中,最直观的容器对象包括: Form、Frame 和 Datamodule。我们可以可视化地将一些组件对象拖放在这些容器对象上。特别是 Frame 更像是专门盛放组件的容器。

Frame 虽然在实际工作时比较像 TPanel,比如一个窗体可以包含多个 Frame,但 Frame 却有着与 Form 一样的可视化继承方法。它和 Form 对包含于其中的组件的实例创建和释放使用了同样的宿主机制,这就是说它可以像窗体那样自动管理包容其中的组件对象。

但在某些方面看来,Frame 更像是一个定制的组件。它可以作为组件保存在组件板上,便于以后的重用。另外,它可以嵌套在窗体、其他 Frame 或其他容器性对象中。在 Frame 被创建并保存后,它仍然会继承其包含的组件 (包括其他 Frame) 发生的改变。当 Frame 被嵌入到其他 Frame 或窗体之中时,它也会继承作为模板的框架发生的变化。因此,Frame 与其说是一个窗口,不如说是一个定制的组件。

Frame 可以更加有效地重用资源。例如,有一幅图像在程序的多个窗体中都要使用,如果采用普通的方法,则在窗体上放置的每一个 TImage 对象都会在资源文件中添加同样的一个图

像副本。而如果将图像加载到 Image 组件中,定制好 Frame,然后在要使用图形的项目中使用定制的 Frame,就会大大缩小窗体文件的大小。Frame 在对程序中多次使用的控件进行组织时也十分有用。例如,用户想在不同的项目中重复使用一组组件(如一套人事资料的编辑框,包括姓名、年龄、职务等),可以先将这些组建添加到 Frame 中定制好 Frame,然后在不同的项目中分别引用定制好的 Frame。这样做的好处在于,以后由于某种需要对原定制 Frame 中的组件的安排进行了修改,则只须重新编译项目,引用 Frame 中的项目将会自动反映出 Frame 的修改。当然,如果直接在引用定制 Frame 的项目中对 Frame 进行修改,则其他引用定制 Frame 的项目将不会反映这种修改。

参见 关于 Frame 的实际应用示例可以参见我著的《Delphi 6 企业级解决方案及应用剖析》第 508 页(机械工业出版社 2002 年出版)

不少程序员在使用容器对象时,基本上是停留在拖放控件的初级应用上。但我们为了提高程序的运行效率和代码的重用,往往会采取动态创建容器及其包容对象的技术,这就要求我们从面向对象的高度对容器对象的使用有更加深入的理解。这里面有些关键的技术问题是我们需要解决的,比如:何时创建容器对象?如何管理容器对象?如何在容器中动态创建组件对象?如何在容器中找到需要的组件对象?

下面我就通过一个动态创建数据模块及其包容对象的例子来演示和介绍容器对象的高级编程技术。

大家都知道,初学 Delphi 的人不习惯使用数据模块(Datamodule)。他们喜欢把数据库组件直接放置在窗体上,觉得这样比较省事。但实际上,由于这么做把业务逻辑和数据处理也留在了窗体之中,使得界面和业务混淆不分,导致程序在以后的维护和修改时非常麻烦。

使用数据模块的好处就是将与数据库存取有关的组件(如: TDataSource、TQuery 等)和处理从界面中分离出来。而 TDatamodule 则是 TDataSource、TQuery 等数据库组件对象最好的容器。

但是对于比较复杂的数据库应用程序,如果把所有窗体中可能用到的数据库存取组件都集中放置在一个公用的数据模块中,则可能会出现一个数据模块中有多达近百个数据库存取组件(我就见过有人在一个数据模块中放置了 50 个 TQuery)。如果在程序设计时,就向数据模块这个容器中放进大量的组件对象,则到实际运行程序时,这些组件的利用效率并不是太高。这是因为程序中有一些不常使用的窗体,与它有关的很多数据库组件(比如 TDataSource、TQuery)也随着公共数据模块一同被创建,占用了大量的资源,造成使用效率极低。

也有人尝试在程序设计时为每个窗体创建一个数据模块,专门放置该窗体需要用到的数据库组件。这样一来,窗体和专用数据模块建立了一一对应的关系,仅仅在该窗体创建时才一起创建属于该窗体的数据模块,这样就不会出现大量闲置的数据库组件对象。

这种办法的确有了改进,但是窗体私有的数据模块中如果存在需要公用的数据库组件怎么办?会不会在很多窗体私有的数据模块中都出现同样的 TDatabase 和 TQuery,同样的组件事件代码?这种冗余同样会降低效率。

最佳的办法就是根据不同的窗体动态创建其私有数据模块,并在数据模块中动态创建数据库组件,而不是在设计时创建数据模块和数据库组件。因为在设计时,我们无法知道用户调用

哪个窗体，使用哪些数据库组件。所以，只有根据运行时用户的操作来动态创建窗体及其私有的数据模块，动态为数据模块增加需要的数据库组件，才能做到量身订做，避免冗余。这样的程序才占用最少的资源，具有最高的效率。

全动态创建的数据库应用程序当然好，可是如何去实现呢？图 4-17 是我提供的示例程序设计方案，图 4-18 是这个设计的 UML 类图。

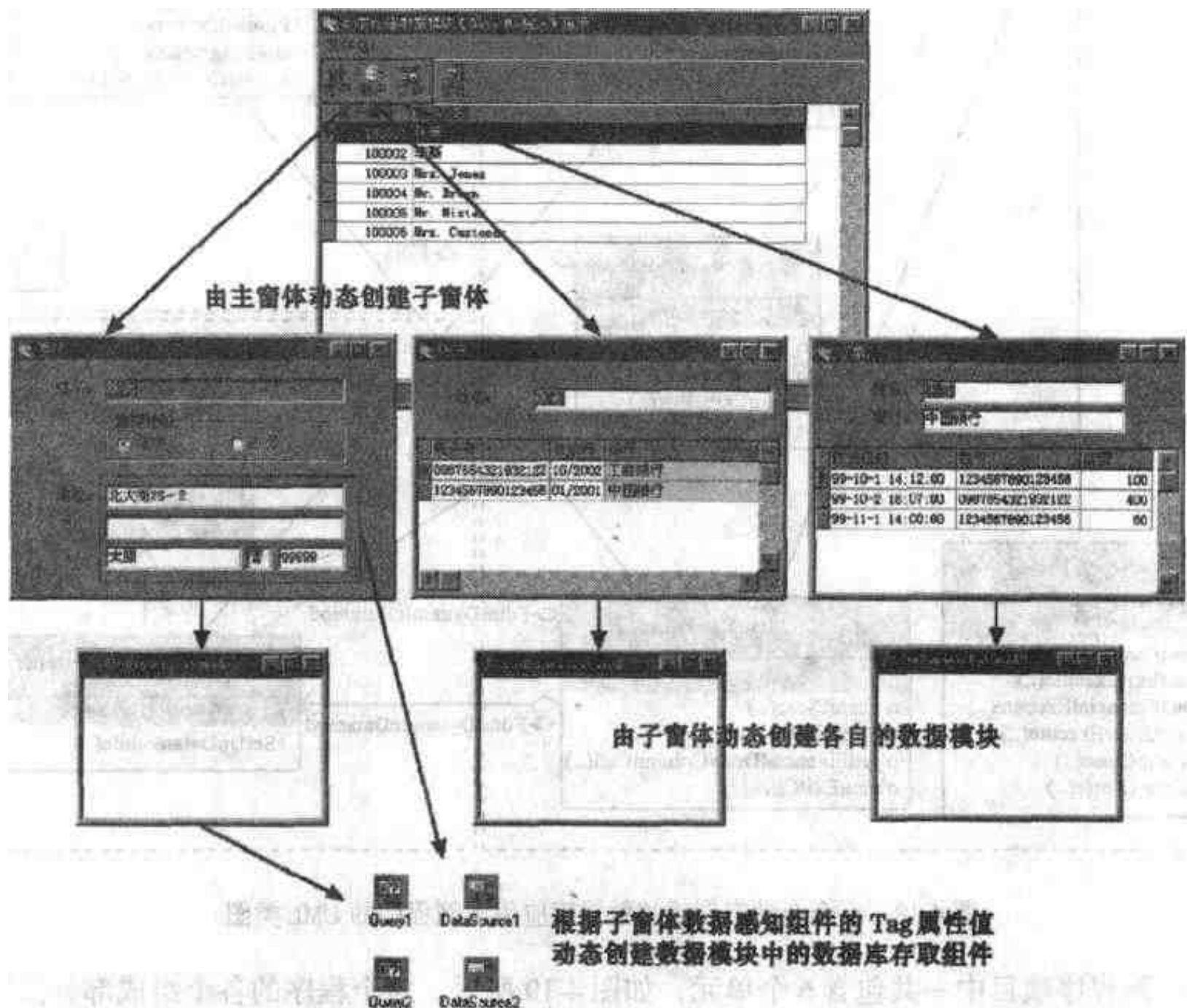


图 4-17 一个全动态创建的数据库应用示例程序设计方案

这个示例程序的设计目标是：

- 每一个窗体的示例都有自己的私有数据模块。该数据模块作为窗体类的数据成员存在。
- 根据窗体的需要，为窗体的数据模块动态创建和加入数据存取组件对象（TDataSource、TQuery）。
- 将业务逻辑代码从窗体和数据模块中抽离出来，由业务对象 TBiz 统一维护。TBiz 作为窗体类的私有数据成员存在。

这样的设计目标既保证了界面、业务和数据对象的分离，又实现了动态创建数据模块的要求。

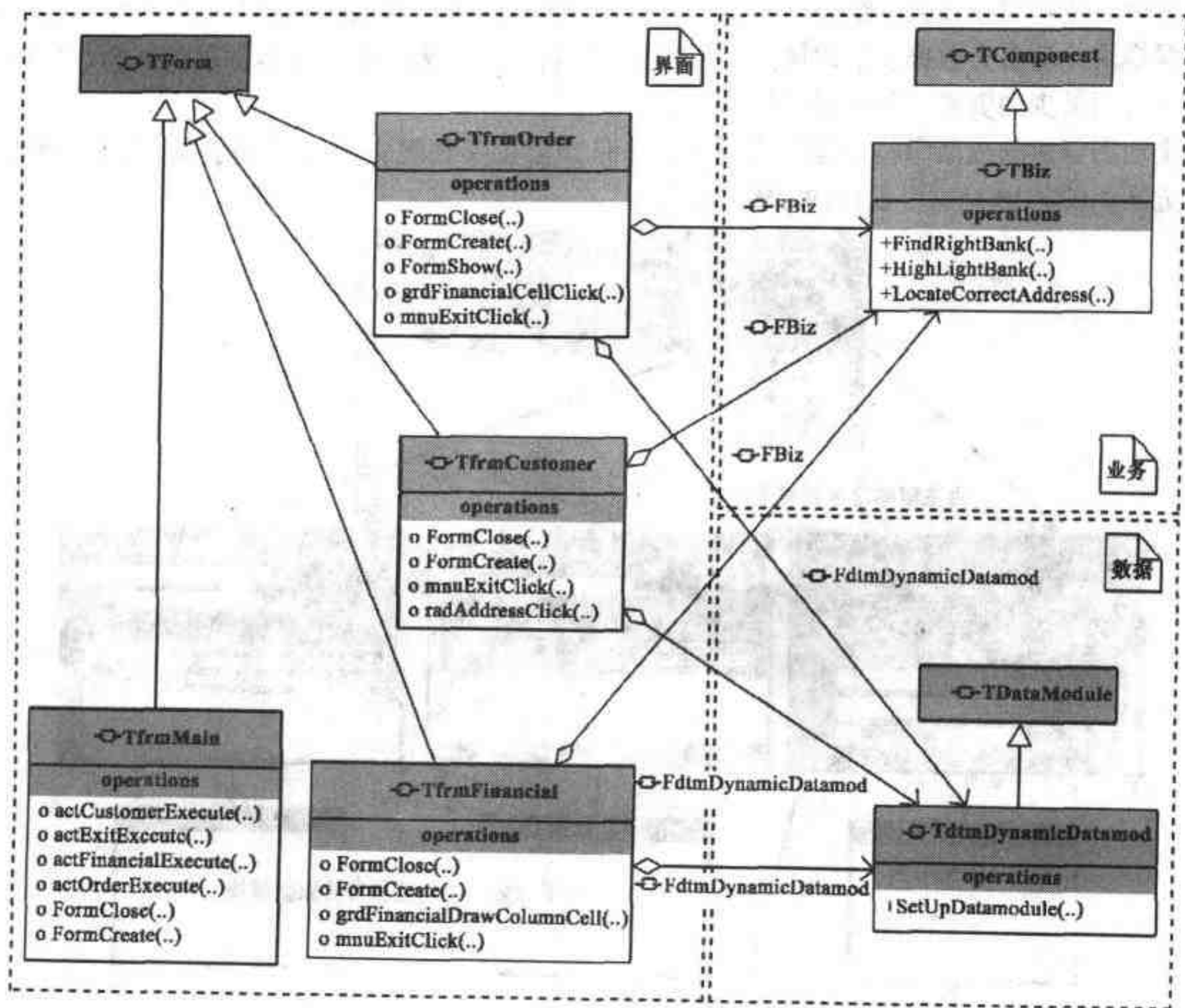


图 4-18 一个全动态创建的数据库应用示例程序的 UML 类图

在示例程序项目中一共包含 6 个单元，如图 4-19 所示。整个程序的各个组成部分及其关系如下所示。

界面部分：

- TfrmMain 类（MainForm 单元）——主窗体对象，用户可以通过该界面选择打开查询数据库明细信息的子窗体。
- TfrmCustomer 类（Customer 单元）——显示客户明细信息的子窗体。
- TfrmFinancial 类（Financial 单元）——显示客户账户明细信息的子窗体。
- TfrmOrder 类（Order 单元）——显示客户订单明细信息的子窗体。

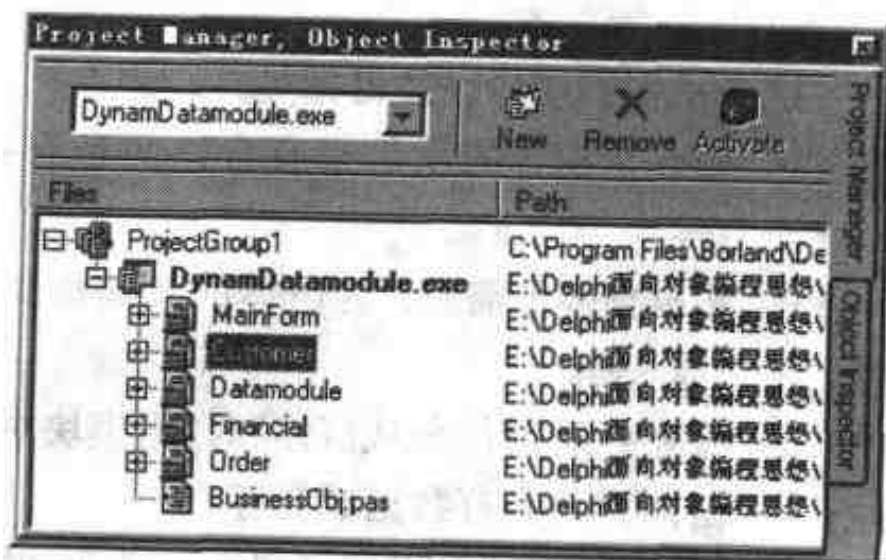


图 4-19 示例程序项目中一共包含 6 个单元

数据部分:

- TdtmDynamicDatamodule 类——包含空白数据模块以及动态创建数据库组件的方法。

业务部分:

- TBiz 类 (BusinessObj 单元) —— 包含用于查询或处理数据的业务逻辑方法。

数据库表:

- DD_Customer.dbf —— 客户表 (TAG 属性值: 101)
- DD_Address.dbf —— 客户地址表 (TAG 属性值: 102)
- DD_Financial.dbf —— 银行账户表 (TAG 属性值: 103)
- DD_Order.dbf —— 订单表 (TAG 属性值: 104)

示例程序 4-7 是主窗体单元源程序, 我们以创建客户窗体 (TfrmCustomer) 的 actCustomerExecute 方法为例来说明动态创建的技巧。

示例程序 4-7 主窗体单元源程序

```
unit MainForm;  
  
interface  
  
uses  
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
  Buttons, Menus, ToolWin, ComCtrls, Grids, DBGrids, ExtCtrls, StdCtrls,  
  DBTables, Db, ActnList, ImgList, StdActns, DBActns;  
  
type  
  TfrmMain = class (TForm)  
    mnuMain: TMainMenu;  
    mnuFile: TMenuItem;  
    mnuExit: TMenuItem;  
    mnuDisplay: TMenuItem;  
    pnlMain: TPanel;  
    grdMain: TDBGrid;  
    qryEntity: TQuery;  
    dtsEntity: TDataSource;  
    ActionList1: TActionList;  
    ImageList1: TImageList;  
    actCustomer: TAction;  
    actFinancial: TAction;  
    actOrder: TAction;  
    ToolBar1: TToolBar;  
    ToolButton1: TToolButton;  
    ToolButton2: TToolButton;  
    ToolButton3: TToolButton;  
    ToolButton4: TToolButton;  
    N1: TMenuItem;  
    N2: TMenuItem;  
    ToolButton13: TToolButton;  
    actExit: TAction;  
    StatusBar1: TStatusBar;  
    procedure FormCreate (Sender: TObject);
```



```

    procedure FormClose (Sender: TObject; var Action: TCloseAction);
    procedure actCustomerExecute (Sender: TObject);
    procedure actFinancialExecute (Sender: TObject);
    procedure actOrderExecute (Sender: TObject);
    procedure actExitExecute (Sender: TObject);
private
    {Private declarations }
public
    {Public declarations }
end;

var
    frmMain: TfrmMain;

implementation

uses Customer, Financial, Order;

{$R *.DFM}

procedure TfrmMain.FormCreate (Sender: TObject);
begin
    qryEntity.open;
end;

procedure TfrmMain.FormClose (Sender: TObject;
    var Action: TCloseAction);
begin
    qryEntity.close;
end;

procedure TfrmMain.actCustomerExecute (Sender: TObject);
var
    vfrmCustomer: tfrmCustomer;
begin
    {客户窗体需要用到客户和地址表。
    创建窗体后,动态创建一个该窗体实例专用的 datamodule。
    并把当前客户编号通过参数传送给 datamodule,以定位客户记录。}
    vfrmCustomer := tfrmCustomer.Create (self);
    vfrmCustomer.FdtmDynamicDatamod.SetUpDatamodule (vfrmCustomer,
        vfrmCustomer.FdtmDynamicDatamod,
        qryEntity [' CustomerNumber' ] );
    vfrmCustomer.Show;
end;

procedure TfrmMain.actFinancialExecute (Sender: TObject);
var vfrmFinancial: tfrmFinancial;
begin
    {账户窗体需要用到客户和账户表。}
    vfrmFinancial := tfrmFinancial.Create (self);
    vfrmFinancial.FdtmDynamicDatamod.SetUpDatamodule (vfrmFinancial,
        vfrmFinancial.FdtmDynamicDatamod,
        qryEntity [' CustomerNumber' ] );
    vfrmFinancial.Show;

```

```

end;

procedure TfrmMain.actOrderExecute (Sender: TObject);
var
    vfrmOrder: tfrmOrder;
begin
    |订单窗体还包含了查询业务|
    vfrmOrder := tfrmOrder.Create (self);
    vfrmOrder.FdtmDynamicDatamod.SetUpDatamodule (vfrmOrder,
        vfrmOrder.FdtmDynamicDatamod,
        qryEntity ['CustomerNumber'] );
    vfrmOrder.Show;
end;

procedure TfrmMain.actExitExecute (Sender: TObject);
begin
    Close;
end;

end.

```

在 actCustomerExecute 方法中首先创建了客户窗体。创建客户窗体时，触发了 TfrmCustomer 的 OnCreate 事件，调用了其中的数据模块创建方法。示例程序 4-8 是客户窗体单元的源程序。

```

procedure TfrmCustomer.FormCreate (Sender: TObject);
begin
    FdtmDynamicDatamod := TdtmDynamicDatamod.Create (self);
    FBiz := TBiz.Create (self);
end;

```

此时随客户窗体创建的是一个空白的数据模块 FdtmDynamicDatamod，以及业务对象 FBiz。它们都是客户窗体的从属对象，由客户窗体来管理其生命期。接着 actCustomerExecute 方法动态创建了一个数据模块（DataModule），并调用该数据模块的 SetUpDatamodule 方法，动态创建其包含的数据库组件。并把当前客户编号通过参数传送给 datamodule，以定位客户记录。这样显示出的客户窗体就有了客户明细信息。

示例程序 4-8 客户窗体单元源程序

```

unit Customer;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls, Mask, DBCtrls, ExtCtrls, Menus, ToolWin, ComCtrls,
    Datamodule, BusinessObj, dbtables;

type
    TfrmCustomer = class (TForm)
        pnlCustomer: TPanel;
        edtCustomerName: TDBEdit;
        edtAddress1: TDBEdit;
    end;

```



```

    edtAddress2: TDBEdit;
    edtAddressCity: TDBEdit;
    edtAddressState: TDBEdit;
    edtAddressZip: TDBEdit;
    radAddress: TRadioGroup;
    lbl1: TLabel;
    lbl2: TLabel;
    procedure mnuExitClick (Sender: TObject);
    procedure FormClose (Sender: TObject; var Action: TCloseAction);
    procedure FormCreate (Sender: TObject);
    procedure radAddressClick (Sender: TObject);
private
    FBiz: TBiz;
public
    FdtmDynamicDatamod: TdtmDynamicDatamod;
end;

implementation

{$R * .DFM}

procedure TfrmCustomer.mnuExitClick (Sender: TObject);
begin
    close;
end;

procedure TfrmCustomer.FormClose (Sender: TObject;
    var Action: TCloseAction);
begin
    Action := caFree;
end;

procedure TfrmCustomer.FormCreate (Sender: TObject);
begin
    FdtmDynamicDatamod := TdtmDynamicDatamod.Create (self);
    FBiz := TBiz.Create (self);
end;

procedure TfrmCustomer.radAddressClick (Sender: TObject);
begin
    FBiz.LocateCorrectAddress (FdtmDynamicDatamod.qryAddress,
        radAddress.Items [radAddress.Itemindex]);
end;

end.

```

动态创建数据模块实例的方法类似于动态创建一个窗体，并不是很难。但动态为数据模块添加组件就不是那么简单了。这就涉及到如何使用容器对象并动态添加和设置其包容的组件对象的技术。示例程序 4-9 是动态数据模块单元的源程序。数据模块的 `SetUpDatamodule` 方法是动态添加和设置数据库组件的关键。

示例程序 4-9 动态数据模块单元源程序

```

unit Datamodule;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBTables, Db, DBCtrls, dbgrids;

type
  TdtmDynamicDatamod = class (TDataModule)
  private
    {Private declarations}
  public
    {These public declarations are actually created within the
    pmDatamoduleManager. Only the ones used by the matching form will be
    created for the instance.}
    datMain: tDatabase;
    qryCustomer: tQuery;
    qryFinancial: tQuery;
    qryAddress: tQuery;
    qryOrder: tQuery;
    dtsCustomer: tDatasource;
    dtsFinancial: tDatasource;
    dtsAddress: tDatasource;
    dtsOrder: tDatasource;
    procedure SetUpDatamodule (const vfrmForm: TForm;
                               var vdtmDynamicDatamod: TdtmDynamicDatamod;
                               const vintCustomer: integer);

  end;

var
  dtmDynamicDatamod: TdtmDynamicDatamod;

implementation

{$R *.DFM}
var vintDatamoduleCount: integer;

procedure TdtmDynamicDatamod.SetUpDatamodule (const vfrmForm: TForm;
                                              var vdtmDynamicDatamod: TdtmDynamicDatamod;
                                              const vintCustomer: integer);

var
  vintLoop2: integer;
  |子过程|
  procedure CreateDataSetPair (vqryCreate: tquery;
                              vdtsCreate: tdatasource;
                              vstrSQL: string;
                              vintTag: integer);

  var
    vintLoop: integer;
  begin
    |创建 query 组件|

```

```

vqryCreate.databasesname := vdtmDynamicDatamod.datMain.databasesname;
vqryCreate.SQL.text := vstrSQL;
vqryCreate.active := true;
|创建连接 query 组件的 datasource 组件|
vdtsCreate.dataset := vqryCreate;
|循环检查 form 上数据库组件的 TAG fields,以便动态关联动态数据模块的 datasource|
For vintLoop := 0 to vfrmForm.ComponentCount - 1 do
|检查数据库感知组件的 tag 属性值是否匹配当前 datasource 的 tag 属性值|
If vfrmForm.Components[vintLoop].tag = vintTag then
|检查 Dbedit 组件,为 Dbedit 组件的 datasource 属性动态赋值|
If vfrmForm.Components[vintLoop] is TDBEdit then
TDBEdit(vfrmForm.Components[vintLoop]).datasource := vdtsCreate
else
|检查 dbGrid 组件,为 dbGrid 组件的 datasource 属性动态赋值|
If vfrmForm.Components[vintLoop] is tdbGrid then
tdbgrid(vfrmForm.Components[vintLoop]).datasource :=
vdtsCreate;
end;
|子过程结束|
|----- SetUpDatamodule 方法为空数据模块动态创建和加入数据库存取组件 -----|
begin
|所有 datamodules 都有 TDatabase 实例|
vdtmDynamicDatamod.datMain := TDatabase.create(vdtmDynamicDatamod);
|.... database 属性设置 ...|
vdtmDynamicDatamod.datMain.aliasname := 'DD_Test';
vdtmDynamicDatamod.datMain.name := 'datDatabase';
inc(vintDatamoduleCount);
|动态创建的多个数据模块中,要解决 TDatabase 命名重复的问题,以免在程序中产生冲突|
vdtmDynamicDatamod.datMain.databasesname := 'Database'
+ inttostr(vintDatamoduleCount);
|所有的 query 组件和它们的 datasource 组件是根据数据库控件中的 tag 属性值来创建的。
tag 值通过编码暗藏了数据库表和字段的信息|
For vintLoop2 := 0 to vfrmForm.ComponentCount - 1 do
|如果 tag 值为 101, 创建客户查询|
If (vfrmForm.Components[vintLoop2].tag = 101) and
(vdtmDynamicDatamod.gryCustomer = nil) then
begin
vdtmDynamicDatamod.gryCustomer :=
Tquery.Create(vdtmDynamicDatamod);
vdtmDynamicDatamod.dtsCustomer :=
Tdatasource.Create(vdtmDynamicDatamod);
CreateDataSetPair(vdtmDynamicDatamod.gryCustomer,
vdtmDynamicDatamod.dtsCustomer,
'Select * from DD_Customer where CustomerNumber = '
+ inttostr(vintCustomer), 101);
end
else
|如果 tag 值为 102, 创建地址查询|
If (vfrmForm.Components[vintLoop2].tag = 102) and
(vdtmDynamicDatamod.gryAddress = nil) then
begin
vdtmDynamicDatamod.gryAddress := Tquery.Create(vdtmDynamicDatamod);
vdtmDynamicDatamod.dtsAddress :=
Tdatasource.Create(vdtmDynamicDatamod);

```

```

CreateDataSetPair (vdtmDynamicDatamod.qryAddress,
                  vdtmDynamicDatamod.dtsAddress,
                  'Select * from DD_Address where CustomerNumber = '
                  + inttostr (vintCustomer), 102 );

end
else
|如果 tag 值为 103, 创建账户查询|
If (vfrmForm.Components [vintLoop2] .tag = 103) and
  (vdtmDynamicDatamod.qryFinancial = nil) then
begin
  vdtmDynamicDatamod.qryFinancial: =
    Tquery.Create (vdtmDynamicDatamod);
  vdtmDynamicDatamod.dtsFinancial: =
    Tdatasource.Create (vdtmDynamicDatamod);
  CreateDataSetPair (vdtmDynamicDatamod.qryFinancial,
                    vdtmDynamicDatamod.dtsFinancial,
                    'Select * from DD_Financial where CustomerNumber = '
                    + inttostr (vintCustomer), 103 );

end
else
|如果 tag 值为 104, 创建订单查询|
If (vfrmForm.Components [vintLoop2] .tag = 104) and
  (vdtmDynamicDatamod.qryOrder = nil) then
begin
  vdtmDynamicDatamod.qryOrder: = Tquery.Create (vdtmDynamicDatamod);
  vdtmDynamicDatamod.dtsOrder: =
    Tdatasource.Create (vdtmDynamicDatamod);
  CreateDataSetPair (vdtmDynamicDatamod.qryOrder,
                    vdtmDynamicDatamod.dtsOrder,
                    'Select * from DD_Order where CustomerNumber = '
                    + inttostr (vintCustomer), 104 );

end;
end;

end.

```

SetUpDatamodule 方法是这样工作的:

首先为数据模块创建 TDatabase 组件对象实例, 并设置好该对象的属性。因为使用 BDE 数据引擎, 故所有数据模块都需要有 TDatabase 组件。

然后根据传递来的窗体对象引用, 依次检查该容器对象上的所有组件的 Tag 属性值。如果 tag 值为 101 就创建客户查询组件 qryCustomer 的对象实例; 如果 tag 值为 102 就创建地址查询组件 qryAddress 的对象实例; 如果 tag 值为 103 就创建账户查询组件 qryFinancial 的对象实例; 如果 tag 值为 104 就创建订单查询组件 dtsOrder 的对象实例。这里的 tag 值是在窗体中的数据感知控件中预设好的。比如: 客户窗体的地址栏组件 edtAddress1 的 tag 值为 102, 对应数据集就为 qryAddress。同样, 数据模块中还创建了连接 query 组件和数据感知控件的 datasource 组件, 并将其设置到客户窗体上对应的数据库感知组件的 datasource 属性中。

对于容器中的所有组件, 我们在程序中使用了容器的对象数组:

```
Components [Index: Integer]: TComponent read GetComponent;
```

这样才能方便地找到需要的组件,进行相关属性值的比较和判定。

所有的明细信息显示窗体还有各自的查询和操作,这些业务逻辑封装在 TBiz 对象中,如示例程序 4-10 所示。比如客户窗体中有一个切换住宅和办公地址的操作,就是调用了 TBiz 的 LocateCorrectAddress 方法。业务逻辑的分离能够适应业务需求变化所造成的改动,使得程序更有弹性。

示例程序 4-10 业务逻辑单元 BusinessObj 的源程序

```
unit BusinessObj;
{本单元基于特定窗体及其数据模块的业务逻辑代码,用于处理相关查询}

interface
  uses dbtables, db, dbgrids, Graphics, variants, Classes;

type
  TBiz = class (TComponent)
  private
    {Private declarations }
  public
    Procedure LocateCorrectAddress (var vqryAddress: Tquery;
                                   const cstrType: string);
    Procedure HighLightBank (var vcolCurrent: TColumn;
                             const cintColumn: integer);
    Procedure FindRightBank (var vqryOrder, vqryFinancial: tQuery);
  end;

implementation

Procedure TBiz.LocateCorrectAddress (var vqryAddress: Tquery;
                                     const cstrType: string);
begin
  vqryAddress.Locate (' CustomerNumber; AddressType',
                     VarArrayOf ( [vqryAddress [' CustomerNumber'], cstrType]),
                     [] );
end;

{为某些含有特殊字符的银行名称进行着色处理,突出显示效果}
Procedure TBiz.HighLightBank (var vcolCurrent: TColumn; const cintColumn: integer);
begin
  If cintColumn = 2 then
    If pos ('中国', vcolCurrent.Field.asstring) > 0 then
      vcolCurrent.Color := clAqua
    else
      If pos (' USA', vcolCurrent.Field.asstring) > 0 then
        vcolCurrent.Color := clRed;
  end;

Procedure TBiz.FindRightBank (var vqryOrder, vqryFinancial: tQuery);
begin
  {根据账户号定位银行}
  vqryFinancial.Locate (' CustomerNumber; CreditCard',
                       VarArrayOf ( [vqryOrder [' CustomerNumber'],
```

```

                vqryOrder ['CreditCard'])),
            [] );
end;

end.

```

由于我们使用了 BDE 数据引擎, 因此, 为了避免在程序发布后手工配置 BDE 的麻烦, 我们在主程序中增加了 BDE 的动态配置代码。动态配置 BDE 不仅是为了方便程序运行, 更是动态创建数据模块及其数据库存取组件的需要。鉴于这方面的资料不多, 下面给出了源代码, 如示例程序 4-11 所示。

示例程序 4-11 示例程序的主程序源代码

```

program DynamDatamodule;
{
Tables:
    (101 in TAG field) DD_Customer - 客户表
    (102 in TAG field) DD_Address - 客户地址表
    (103 in TAG field) DD_Financial - 银行账户表
    (104 in TAG field) DD_Order - 订单表
}
uses
    Forms,
    bde,
    dbtables,
    MainForm in 'MainForm.pas' {frmMain},
    Customer in 'Customer.pas' {frmCustomer},
    Datamodule in 'Datamodule.pas' {dtmDynamicDatamod: TDataModule},
    prmbusinesscode in 'prmbusinesscode.pas',
    Financial in 'Financial.pas' {frmFinancial},
    Order in 'Order.pas' {frmOrder};

{$R *.RES}

begin
    Application.Initialize;
    {动态设置 BDE, 为数据库设置别名、路径}
    bde.DbiInit (nil);
    Session.ConfigMode := cmSession;
    Session.AddStandardAlias ('DD_Test',
        '..',
        'DBASE');
    Session.ConfigMode := cmAll;
    Application.CreateForm (TfrmMain, frmMain);
    Application.Run;
end.

```

通过研究和学习源代码可以提高自己的编程能力, 这一点是毋庸置疑的。有兴趣的读者可以到随书光盘中找到这个示例程序的全部源代码文件及完整的项目。

4.4 使用对象参数

在 Delphi 中, 如果两个变量类型相同或兼容, 可以把其中一个对象变量赋给另一个对象变

量。例如，对象 TForm1 和 TForm2 都是从 TForm 继承下来的类型，而且 Form1 和 Form2 已被声明过，那么你可以把 Form1 赋给 Form2：

```
Form2 := Form1;
```

只要赋值的对象变量是被赋值的对象变量的祖先类型，就可以将一个对象变量赋给另一个对象变量。例如，下面是一个 TDataForm 的类型说明，在变量说明部分一共说明了两个变量：AForm 和 DataForm。

```
type
  TDataForm = class (TForm)
    Button1: TButton;
    Edit1: TEdit;
    DataGrid1: TDataGrid;
    Datasheet1: TDataSource;
    TableSet1: TTableSet;
    VisibleSession1: TVisibleSession;
  private
  public
  end;

var
  AForm: TForm;
  DataForm: TDataForm;
```

因为 TDataForm 是 TForm 类型的后代，所以 Dataform 是 AForm 的后代，因此下面的赋值语句是合法的：

```
AForm := DataForm;
```

这就是对象的转型概念，在面向对象编程中是极为重要的，后面我会在第 5 章中详细讨论。通过转型我们可以更加灵活地利用对象参数来传递对象的引用。

最典型的对象参数是我们经常看到的事件处理过程中的 Sender 参数。下面是一个按钮控件的 OnClick 事件处理过程：

```
procedure TForm1.Button1Click (Sender: TObject);
begin
  {实现代码}
end;
```

为什么 Sender 参数总是 TObject 类型呢？因为 TObject 类在 Delphi 的 VCL 的根部，这就意味着所有的 Delphi 对象都是 TObject 的后代。Sender 是 TObject 类型，所以任何对象都可以赋值给它。虽然无法看见赋值的程序代码，但事实上发生事件的组件（或控件）已经赋给 Sender 了，这就是说 Sender 的值是响应发生事件的组件（或控件）的。

如果我们能够确定作为对象参数 Sender 传递的对象的原来类型，虽然该对象在传递中已经转型，但我们仍然可以利用 as 操作符将其恢复为原来的类型。利用这一技术可以快速简洁地开发出像计算器这样的程序。

不信你就这样操作：在窗体上放一个 Edit 和 10 个 Button，把 Button.Caption 分别设为“0”、“1”、“2”……“9”，然后写一个按钮的 OnClick 事件如下：


```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Edit1.Text := Edit1.Text + (Sender as TButton).Caption;  
end;
```

然后把别的 Button 的 OnClick 事件都关联到 Button1Click 上，运行程序。OK，这样计算器程序的雏形就具备了。我们用 Delphi 的 Sender 参数，可以开发出类似 VB 中控件数组功能的程序，但使用 VB 的人却无法享用 Sender 参数的乐趣。

Sender 作为最常用的对象参数在 Delphi 面向对象编程中有着十分重要的意义。我们使用 VCL 控件编程时经常会遇到这个参数或类似这样的参数（比如后面例子中提到的 source 参数），但很多 Delphi 编程人员并不知道使用对象参数传递对象的奥秘和技巧。在面向对象的思维中，对象参数更深刻的含义在于它实现了对象（控件）之间的通信，这里的赋值实际上是对象引用的传递过程。推而广之，我们在设计自己的对象（即不一定是 VCL 控件）时，也可以使用这种高效的通信机制。

下面我来举一个有意思的例子以进行详细讨论。

假如我们要设计这样一个实现拖放操作的程序，该程序如图 4-20 所示，当用户将左边的 Edit1 和 Label1 拖放到右边的 ListBox1 中时，就在 ListBox1 中添加了一项（Item）文字，内容为 Edit1 和 Label1 显示的内容，即：Edit1.Text 和 Label1.Caption。这里面就有一个如何识别 Edit1 和 Label1 对象的问题。

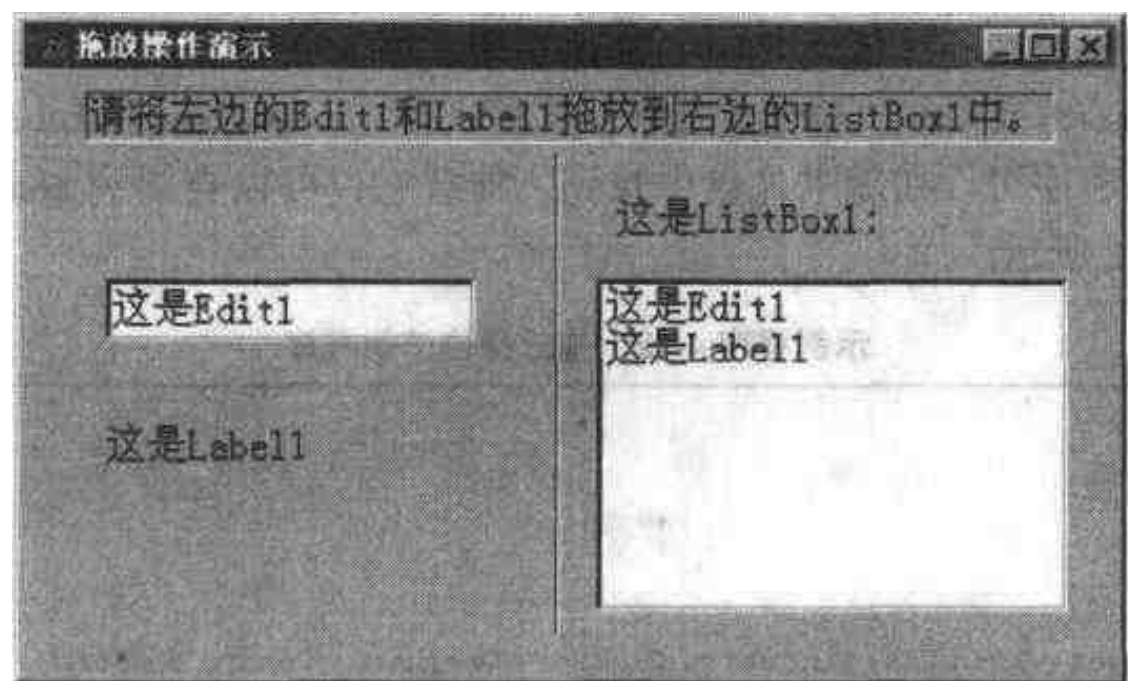


图 4-20 在拖放操作中，需要识别被拖放的对象

这个示例程序的实现步骤如下：

新建项目，在窗体上放置一个 Edit、一个 Label 和一个 ListBox 控件。

1) 为 Edit1 和 Label1 分别建立 OnMouseDown 事件响应。

每一个控件都有一个 DragMode 属性。要实现可停靠控件，需要将 DragMode 属性设为 dmAutomatic。如果将该属性设为 dmManual，则需要调用 BeginDrag 方法才会产生拖放事件。

```
procedure BeginDrag(Immediate: Boolean; Threshold: Integer = -1);
```

当 Immediate 参数为 True 时，拖动操作会立即开始，鼠标指针会变为控件的 DragCursor 属性

指定的指针。如果为 False, 则只有当鼠标指针移动到了超过 Threshold 参数指定的距离时, 拖动操作才会开始。

使用 BeginDrag (False) 方法可以使控件在不产生拖放操作的情况下接受鼠标的单、双击。在这个例子中我们有条件地选择是否开始拖放操作, 在拖放操作开始时检查鼠标是否按左键拖放。并在鼠标左键按下直到拖动超过 10 个像素时才开始拖动操作。

2) 为列表框添加 OnDragOver 事件响应。

当用户拖动一个对象经过某个控件时, 该控件会产生 OnDragOver 事件, 我们可以在该事件中指定是否接受被拖动的对象。接受与否可以通过鼠标指针来表示。

```
type TDragOverEvent = procedure (Sender, Source: TObject; X, Y: Integer; State:
TDragState; var Accept: Boolean) of object;
property OnDragOver: TDragOverEvent;
```

我们可以通过 Accept 参数指定是否接受该拖放对象。X 和 Y 参数表明了当前鼠标指针的位置。在这个例子中, 只有当拖放对象是 Edit1 或 Label1 时, 列表控件才会接受它。

3) 为列表框建立 OnDragDrop 事件响应, 处理拖动对象销毁后的行为。

当控件确认了拖动对象之后, 要着手处理如何接受销毁的对象, 此时可以为 OnDragDrop 事件建立响应句柄。

```
type TDragDropEvent = procedure (Sender, Source: TObject; X, Y: Integer) of object;
property OnDragDrop: TDragDropEvent;
```

与 OnDragOver 事件类似, OnDragDrop 事件也指明了鼠标指针位置等信息。在这个例子中, 列表控件会接受从 Edit1 或 Label1 中拖放过来的文字 (即: Edit1.Text 和 Label1.Caption)。

这个 DragObject 程序可以在随书光盘中找到。加载它, 可以查阅到 MainForm.pas 单元的代码, 如示例程序 4-12 所示。

示例程序 4-12 拖放操作演示程序

```
unit MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class (TForm)
    Edit1: TEdit;
    ListBox1: TListBox;
    Label1: TLabel;
    StaticText1: TStaticText;
    Bevel1: TBevel;
    Label2: TLabel;
    procedure Edit1MouseDown (Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);
    procedure ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;
```

```

        State: TDragState; var Accept: Boolean);
    procedure Label1MouseDown (Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
private
    {Private declarations }
public
    {Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Edit1MouseDown (Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    {只有当鼠标左键按下时 }
    if Button = mbLeft then
        {直到拖动超过 10 个像素时才开始拖动操作 }
        Edit1.BeginDrag (False, 10);
end;

procedure TForm1.Label1MouseDown (Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if Button = mbLeft then
        Label1.BeginDrag (False, 10);
end;

procedure TForm1.ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);
begin
    {判断拖动的对象,在列表框中添加对应的文字 }
    if (Source = Edit1) and (Sender = ListBox1) then
        ListBox1.Items.Add (Edit1.Text);
    if (Source = Label1) and (Sender = ListBox1) then
        ListBox1.Items.Add (Label1.Caption);
end;

procedure TForm1.ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin
    {在被拖动对象是 Edit1 或 Label1 时接受 }
    if (Source = Edit1) or (Source = Label1) then
        begin
            Accept := True;
        end;
end;

end.

```

在上面的例子中, ListBox1 需要知道我们拖放的是哪一个对象, 是 Edit1 还是 Label1? 因为

在 OnDragOver 和 OnDragDrop 事件中 Source 参数传递了拖放对象,为此我们使用了 Source = Edit1 和 Source = Label1 来判断拖放的对象是哪一个,显然我们涉及到了对象本身,即对象实例。这种编程方法没有体现面向对象的思想,因而并不是设计良好的代码,原因如下:

首先, Edit1 和 Label1 作为 TEdit 和 TLabel 类的实例被写死在 TForm1 类的实现中,不利于扩展。比如,现在需要对程序进行扩展,增加一个 Edit2 对象,并实现同样的拖放,此时需要同时修改 OnMouseDown、OnDragOver 和 OnDragDrop 事件中的代码。那么每次增加同一类型的拖放对象都要修改一次代码,岂不是很麻烦,哪里体现了程序的复用性?其次,对于不同类型的控件无法实现多态。从严格意义上来说,也不符合封装性的要求。

下面我将把示例程序 4-12 进行改进,使之更符合面向对象的要求。

首先,我们来分析示例程序 4-12 的 TForm1.Edit1MouseDown 方法:

```
procedure TForm1.Edit1MouseDown (Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    {只有当鼠标左键按下时}  
    if Button = mbLeft then  
        {直到拖动超过 10 个像素时才开始拖动操作}  
        Edit1.BeginDrag (False, 10);  
end;
```

这里的 OnMouseDown 事件是针对具体的拖放对象 Edit1 的。将 Edit1.BeginDrag (False, 10); 写死在方法中使得该方法只能用于 Edit1 对象,而不能用于其他的 Edit 对象,如 Edit2。我们知道,一个具体的 Edit 对象只不过是 TEdit 的实例,TEdit 类封装和抽象了所有 Edit 对象的特征,那么 TEdit 的 BeginDrag 动作肯定适用于所有的 Edit 对象。所以,我们如果使用 OnMouseDown 事件的 Sender 参数来传递不同的 TEdit 类型的对象,那么就可以使所有 Edit 对象共用这段 OnMouseDown 事件代码了。现在我们将针对 Edit1 对象的 OnMouseDown 事件 TForm1.Edit1MouseDown 改为以下针对所有 Edit 对象(即针对 TEdit 类型)的 OnMouseDown 事件 TForm1.EditMouseDown:

```
procedure TForm1.EditMouseDown (Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Integer);  
begin  
    if Button = mbLeft then  
        (sender as TEdit).BeginDrag (False, 10);  
end;
```

注意在上述代码中出现了类型转换。因为 Sender 参数是 TObject 类型, TObject 类型没有 BeginDrag 方法,所以我们要使用类操作符 as 将该类型转换为 TEdit 类型。这样只要是 TEdit 类型的变量就可以使用这段代码,不管它是 Edit1 还是 Edit2。

也许你会问,为什么 Sender 参数是 TObject 类型而不是其他类型的(比如 TEdit 类型,这样就不需要转换类型了)?这是个好问题,试想 Delphi 并不知道你要传递的是何种类型的对象变量,但是 Delphi 知道所有对象的类都源自于 TObject, TObject 是所有类的祖先;于是使用 TObject 类型的 Sender 参数就可以做到以不变应万变,用来传递任何类型的对象。这并不是 Delphi 的神奇,而是面向对象的神奇。在面向过程的编程中,有谁见过这种传递参数的方法

呢?

再进一步,我们能不能写一段 OnMouseDown 事件代码,使之不仅适用于 Edit 对象,也能适用于 Label 对象呢?可能读者已经想到,既然 Sender 参数能传递任何类型的对象,当然也能传递 TLabel 类型的对象,所以我们所要做的就是判断 Sender 参数传递的是何种类型的对象,并进行对应的类型转换。现在我们将针对 TEdit 类型对象的 OnMouseDown 事件 TForm1.EditMouseDown 改为以下针对 TEdit 类型对象和 TLabel 类型对象的 OnMouseDown 事件 TForm1.ControlMouseDown:

```
procedure TForm1.ControlMouseDown (Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);  
begin  
  if Button = mbLeft then  
  begin  
    if (sender is TEdit) then  
      (sender as TEdit).BeginDrag (False, 10);  
    if (sender is TLabel) then  
      (sender as TLabel).BeginDrag (False, 10);  
  end;  
end;
```

在上述代码中,我们使用类操作符 is 来判定 Sender,以便找到调用这个事件处理过程的组件(或控件)的类型。也就是说,通过对象类型来判定拖动的是 Edit1 对象还是 Label1 对象。将原来的单个对象识别提升到对象类型识别的好处是,可以同时处理多个 Edit 对象和 Label 对象,而不必为每个对象都编写单独的识别代码。

分析代码,我们还发现,TEdit 和 TLabel 都有相同的 BeginDrag 方法,这意味着它们可能从共同的祖先类中继承了 BeginDrag 方法。这个共同的祖先类就是 TControl。我们可以将代码最后优化成这样:

```
procedure TForm1.ControlMouseDown (Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);  
begin  
  if Button = mbLeft then  
    (sender as TControl).BeginDrag (False, 10);  
end;
```

这样的代码实际上支持所有可以拖动的控件,你不需要知道这个控件是 TLabel 还是 TEdit,因为它们都是 TControl 类型的派生类,如图 4-21 所示。

要识别一个对象首先是识别该对象的类型,然后再根据该对象的属性进一步确定具体的对象。

现在,我们把针对不同种类控件对象的 OnMouseDown 事件(如:Edit1MouseDown、Label1MouseDown)统统合并为针对所有控件对象的 OnMouseDown 事件。新的 OnMouseDown 事件 ControlMouseDown 是针对所有控件的,也就是说只有将针对具体对象的操作抽象到针对类的操作,将针对特定类(如 TEdit、TLabel)的操作抽象到针对普遍类(如 TControl)的操作时,才能提高代码的复用性。

同样,我们在 ListBox1DragOver 方法中也是接收了对所有控件的拖动,尽管对象参数是 Source 而不是 Sender。

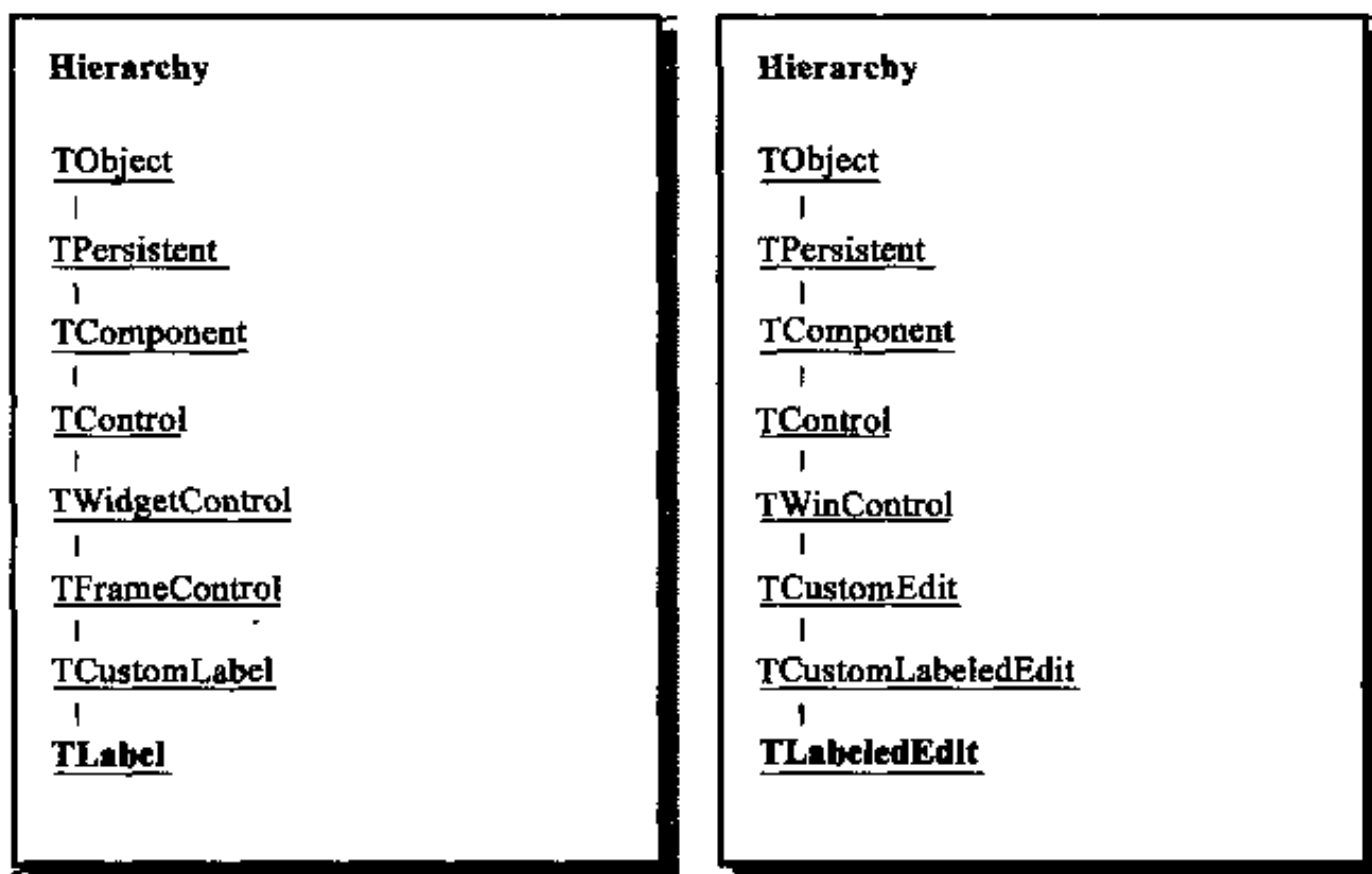


图 4-21 TLabel 的 VCL 继承关系表明它和 TEdit 都有共同的祖先类 TControl

```

procedure TForm1.ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin
    Accept := (Source is TControl);
end;
  
```

Source 参数也是 TObject 类型的，Source 被赋值为那个被拖放的对象。用 ListBox1DragOver 方法的目的是确保只有控件才可以被拖放。Accept 是布尔型参数，如果 Accept 为 True，那么用户选择的控件可以被拖放；反之当 Accept 的值为 False 时，用户就不可以拖放所选择的控件。类操作符 is 检查 Source 是否是 TControl 的类型，所以 Accept 只有在用户拖放一个控件时才为真，并作为变参输出到函数之外。

下面的 ListBox1DragDrop 显示了 OnDragDrop 事件处理过程中也使用了 Source 参数。这个方法是为了在 ListBox1 中加入文字。注意到加入的文字分别来自 TEdit 的 Text 属性和 TLabel 的 Caption 属性，所以需要判断 Source 对象是何种类型，并转换到该类型，这样才能分别找到各自的属性：

```

procedure TForm1.ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);
begin
    if Source is TCustomEdit then
        ListBox1.Items.Add ( (Source as TCustomEdit) .Text );
    if Source is TLabel then
        ListBox1.Items.Add ( (Source as TLabel) .Caption );
end;
  
```

注意，在这段代码中没有用 TEdit 类型，而是用了 TEdit 的基类 TCustomEdit，这是想说明在代码设计时应该尽量做一些“宽口径”的类型考虑。比如这里使用了 TCustomEdit 类型，那么对于新增 TLabeledEdit 控件或用户自定义的第三方 Edit 控件，无需更改代码即可以支持。

TLabeledEdit 和 TEdit 的 VCL 继承关系如图 4-22 所示。

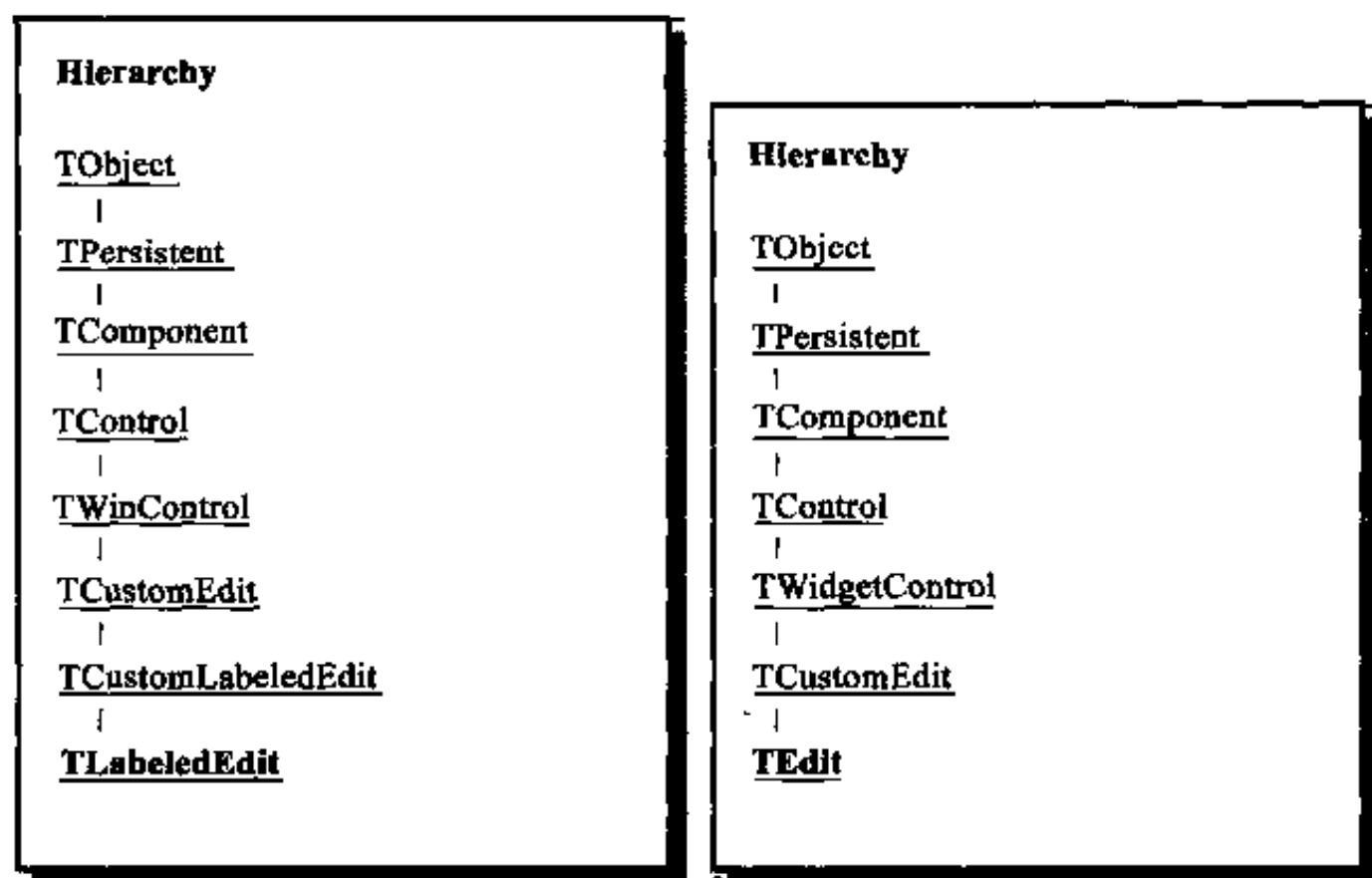


图 4-22 TEdit 和 TLabeledEdit 的 VCL 继承关系表明它们的基类都是 TCustomEdit

在很多情况下，当你为拖放事件处理过程编写赋值语句时，并不知道（也无需知道）用户会拖放哪一个控件、哪一种控件，只有通过类型判断和类型转换才能把正确的对象属性赋给 ListBox1 的 Item。Source 传递了用户拖放对象的引用，虽然 Source 参数定义为 TObject 类型，但它是一个“桥梁”，通过它不同类型的拖放对象向上转型（隐式的，自动由原来的类型转型为 TObject 类型）进行传递，然后再通过向下转型（显式的，使用 as 转回需要的类型）完成和 ListBox1 对象的通信，实现了拖放操作。这里面包含了面向对象的精髓——多态，在后面第 5 章我们将进行更深入的阐述。

最后我们看到的演示程序如图 4-23 所示。这个程序比改进前增加了一个 Edit2 和一个 LabeledEdit1 拖放对象，但代码却更简洁、更灵活、更高效，使我们真正体验到了面向对象的魅力。这个 DragObject 程序可以在随书光盘中找到。它的完整代码如示例程序 4-13 所示。

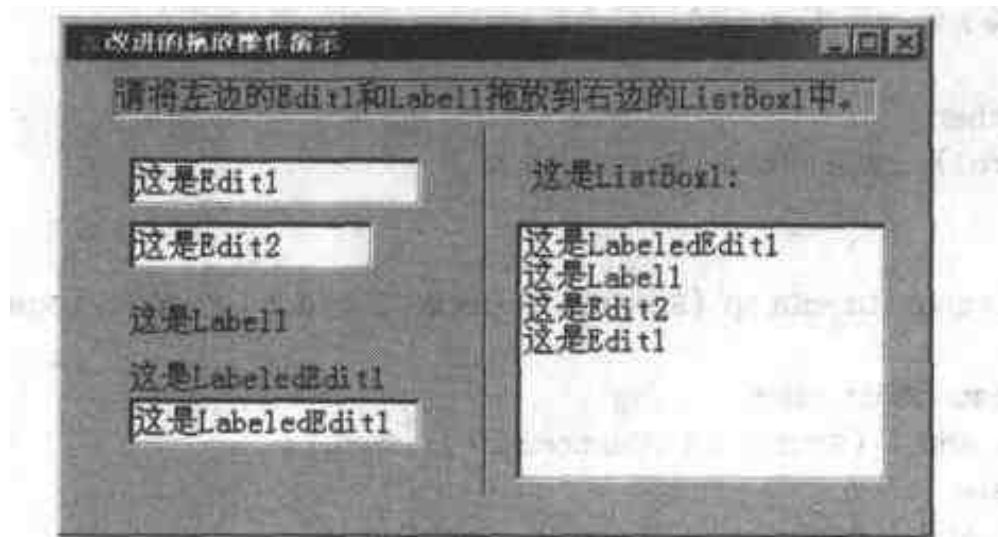


图 4-23 在拖放操作中，通过识别和转换被拖放的对象类型可以优化程序

 示例程序 4-13 改进过的拖放操作演示程序

```

unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class (TForm)
    Edit1: TEdit;
    ListBox1: TListBox;
    Label1: TLabel;
    StaticText1: TStaticText;
    Bevel1: TBevel;
    Label2: TLabel;
    Edit2: TEdit;
    LabeledEdit1: TLabeledEdit;
    procedure ControlMouseDown (Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);
    procedure ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;
      State: TDragState; var Accept: Boolean);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ControlMouseDown (Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then
    (sender as TControl).BeginDrag (False, 10);
end;

:
procedure TForm1.ListBox1DragDrop (Sender, Source: TObject; X, Y: Integer);
begin
  if Source is TCustomEdit then
    ListBox1.Items.Add ( (Source as TCustomEdit).Text);
  if Source is TLabel then
    ListBox1.Items.Add ( (Source as TLabel).Caption);
end;

procedure TForm1.ListBox1DragOver (Sender, Source: TObject; X, Y: Integer;

```

```
State: TDragState; var Accept: Boolean);  
begin  
    Accept: = (Source is TControl);  
end;  
  
end.
```

4.5 组件开发中的面向对象思考

要成功开发组件，关键要认识到组件是类。对于好的类和好的组件来说，两者的开发规范是基本相同的。

开发组件通常需要有高度的面向对象的开发技巧，因为组件实际上就是一个可重用的对象，所以，创建一个组件与创建一个应用程序相比要求编程者更应具备面向对象的编程思想，并进行面向对象的思考。一个优秀的组件往往也体现了面向对象编程思想中的精髓部分，例如封装性、继承性和多态性，以及对象间的合成关系、依赖关系等。

通常创建一个新的组件是在一个已有的组件基础上再创建一个新类，从本质上说也就是通过继承和合成的关系对原有组件的 VCL 库进行扩充。通过继承关系，一些新的功能便能很快在新类中实现；通过合成关系，一些不同的组件便能很快加入到新类中，合成为功能更强的新组件。但是，创建组件与创建应用程序最显著的区别在于，应用程序开发者仅仅是更改组件属性值或者调用组件的方法来编写程序，而组件开发者则是编写代码来定义所设计组件在实际应用当中的行为。

4.5.1 开发 VCL 组件

与在 Delphi 中开发应用程序不同的是创建自定义组件的过程是一个非可视化过程。例如，创建组件并不是基于窗体的，因此窗体设计器不能使用。同样由于对象编辑器只能显示组件在 Delphi 开发环境中已经注册过的信息，因此该可视化工具也不能使用。组件的创建过程全部是通过写 Object Pascal 代码来实现的。虽然在创建组件过程中，可视化设计的特征没有了，但同样的开发环境仍然存在。不仅是代码编辑器，在开发的过程中集成化的调试工具和对象浏览器就已经大大地简化了我们的工作，而且 Delphi 的实时在线帮助也可帮助我们解决实际问题，这更令我们如虎添翼。

要声明的是在两种情况下创建组件需要用到窗体设计器，分别是创建一个对话框组件和创建一个定制的属性或组件编辑器；这两种情况都需要在窗体设计器中设计和创建对话框。此时，你仍然需要编写一些代码来使窗体在适当的时候显示出来。

一般而言，编写一个组件分成 6 个步骤：

- 1) 确定一个祖先类。
- 2) 创建一个组件单元。
- 3) 在新组件中添加属性、方法和事件。
- 4) 测试该组件。
- 5) 在 Delphi 中注册该组件。
- 6) 为该组件建立帮助文件。

图 4-24 是创建组件的流程图。实际上组件单元与普通的 Delphi 单元基本上没有区别。组件单元也有接口部分和实现部分。最重要的差异是实现部分 Register 过程的说明和定义。当使用 Component|Install Component 菜单项安装组件时, Delphi 将执行相应的 Register 过程, 该过程调用了 RegisterComponent 过程。RegisterComponent 过程第一个参数表示把组件添加到组件面板的指定属性页上, 第二个参数表示要注册的组件。

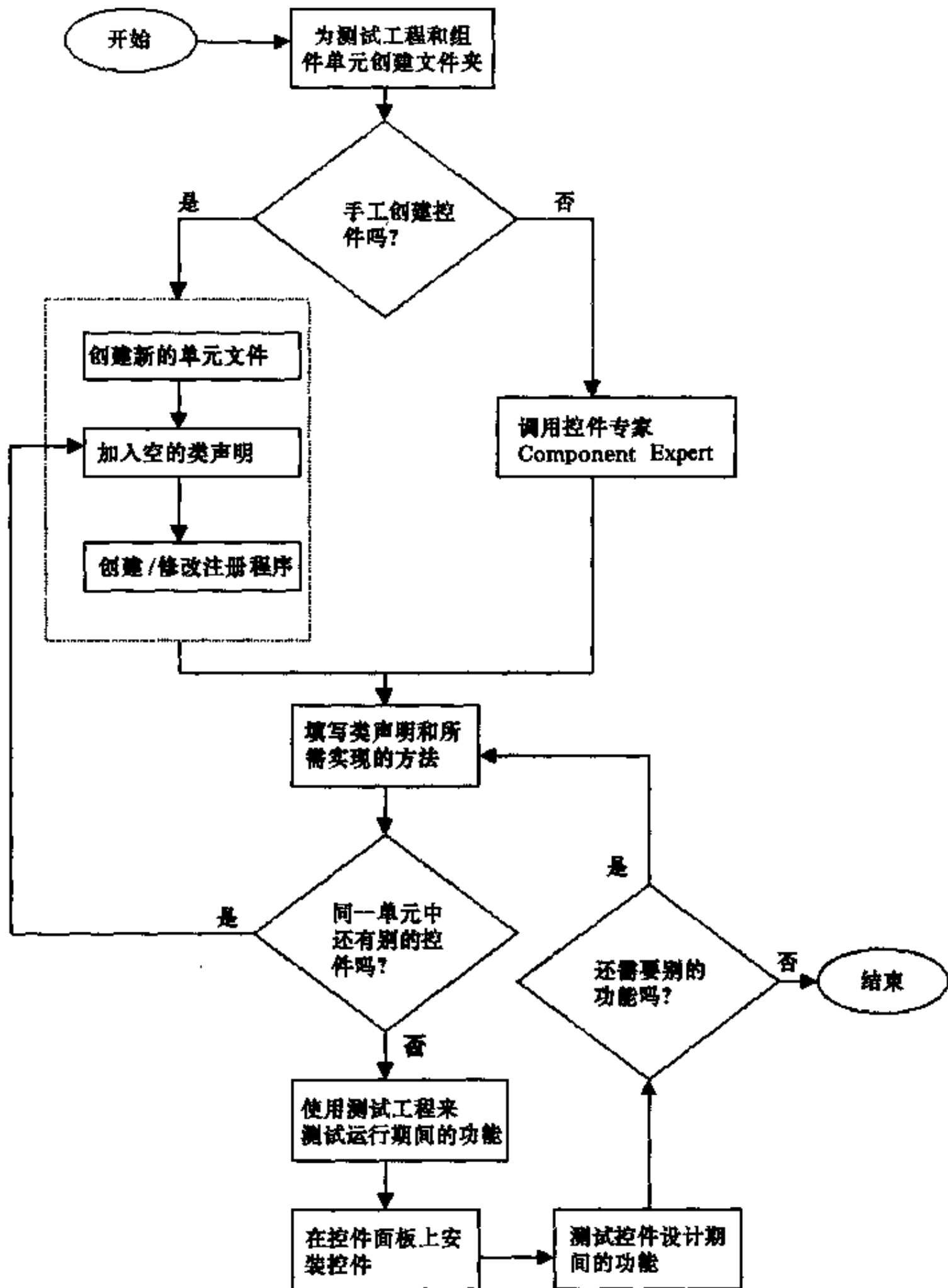


图 4-24 创建组件的流程图

记住，组件开发是一项渐进式的工作。即通过对组件的功能进行增量和迭代式的修改可以最终完善一个优秀实用的组件。所以一开始就在一个组件中包含过多的功能是没有必要的。

实际上，现在在很多 Delphi 的读物中都不乏介绍如何开发 VCL 组件的内容，开发一个简单的 VCL 组件几乎易如反掌，但开发真正有价值的商业组件却并不容易。组件开发的突破之处在于你是否真正站在面向对象的高度，进行了面向对象的深入思考。由于在本书中我想重点讨论组件的几种面向对象开发思路，而不是打算介绍组件开发方面的入门知识，所以读者欲了解如何开发 VCL 组件更详细的方法可以参考其他书籍。

4.5.2 继承

通过继承可以实现对象的重用，继承使我们创建的组件既保留了前组件的功能又实现了许多新功能。继承是组件开发中首先要考虑的面向对象开发方式。一般有以下几种思路。

1. 公布或隐藏继承组件的属性/事件

通过公布或隐藏继承组件的属性/事件可以快速创建一个自己需要的新组件。Delphi 在设计自己的 VCL 组件时就是采用的这种方法。在 Delphi 的 VCL 中，我们可能会发现大量的 TCustomXXX 类，比如：

TCustomCheckBox	TCustomListView
TCustomComboBox	TCustomMaskEdit
TCustomControl	TCustomMemo
TCustomDBGrid	TCustomOutline
TCustomEdit	TCustomPanel
TCustomGrid	TCustomRadioGroup
TCustomGroupBox	TCustomRichEdit
TCustomHotKey	TCustomTabControl
TCustomImageList	TCustomTreeView
TCustomLabel	TCustomUpDown
TCustomListBox	

这些类的所有属性都是被保护的 (protected)，无法在 Object Inspector 中看到。它们不出现在组件面板上，而是专门用于创建其他组件。比如：TCustomMaskEdit、TCustomMemo、TDBLookupCombo、TEdit 和 TSpinEdit 都是由基类 TCustomEdit 派生的，并且通过继承 TCustomMaskEdit 又创建了 TDBEdit 和 TMaskEdit 组件。

通过将被保护的属性/事件有选择地公布出来，使之成为公布的 (published)、可以在 Object Inspector 中修改的属性/事件不失为一种简单有效地创建新组件的思路。例如，TSpeedButton 组件没有 Align 属性用于位置对齐，通过继承我们将原来受保护的 Align 属性公布出来，通过示例程序 4-14 创建了一个有按钮位置对齐功能的新组件，称之为 TAligningSpeedButton 组件。

示例程序 4-14 有按钮位置对齐功能的 TAligningSpeedButton 组件

```
unit AlignBtn; {TAligningSpeedButton 组件}

interface
```

```

uses
  Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, Buttons;

type
  TAligningSpeedButton = class (TSpeedButton)
  published
    property Align; {公布继承的属性}
  end; {TAligningSpeedButton}

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents ('New', [TAligningSpeedButton]);
end; {Register}

end.

```

通过 VCL 类的继承关系知道，从 TControl 开始，组件就继承了受保护的 Align 属性，我们在 TAligningSpeedButton 组件中公布了该属性，使得可以在 Object Inspector 中可视化使用新组件的位置对齐功能。

```

TObject
  TPersistent
    TComponent
      TControl {保护的 Align 属性}
        TGraphicControl
          TSpeedButton
            TAligningSpeedButton {公布的 Align 属性}

```

同样，我们也可以通过隐藏一些不需要用户在 Object Inspector 中修改的属性来创建新组件。简单的方法是在派生的新组件中用相同的名称重新声明该属性，并使其只读。示例程序 4-15 就通过这种方法隐藏了组件的 Left、Top、Height 和 Width 属性。

示例程序 4-15 隐藏了 Left、Top、Height 和 Width 属性的 TSnapPanel 组件

```

TSnapPanel = class (TPanel)
private
  {Private declarations}
  FDummyProperty: byte; {数据成员 Dummy 用于隐藏属性}
  .....
published
  {Published properties and events}
  property Height: byte read FDummyProperty; {隐藏属性}
  property Left: byte read FDummyProperty; {隐藏属性}
  property Top: byte read FDummyProperty; {隐藏属性}
  property Width: byte read FDummyProperty; {隐藏属性}
end; {TNumberEdit}

```

如果需要访问组件中已经隐藏的属性则可以使用关键字 `inherited`:

```
procedure TSnapPanel.SnapUpperLeft;  
begin  
    inherited Left := BorderLeft;  
    inherited Top := inherited Left;  
end; {SnapUpperLeft }
```

2. 覆盖触发事件的虚方法

通过覆盖基类的方法可以快速扩展基类的功能。随着不断使用 Delphi 的组件，我们很快就能对每个组件中不同的事件有所了解。当新建一个派生组件时，初学者通常会犯这样的错误，他们在派生组件的构造方法中动态创建和放置事件句柄。这种方法会造成当组件的使用者为该句柄分配事件时，事件句柄无法被调用。更好的挂接事件的方法是，覆盖那个响应你要处理的事件的虚方法，特别当该虚方法存在时更应如此。

为了找到虚方法及其对应的事件，需要检查代码。通常在 VCL 中，`OnXXX` 事件是由 `XXX` 虚方法触发的。比如：组件的 `OnClick` 事件是由 `Click` 虚方法触发的。示例程序 4-16 演示了 `TButton` 的派生组件 `TWaveButton` 通过覆盖触发 `OnClick` 事件的 `Click` 虚方法来实现播放声音的按钮功能。

示例程序 4-16 能播放声音的按钮组件 `TWaveButton`

```
unit WavBtn; {TWaveButton component.}  
  
interface  
  
uses  
    Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, StdCtrls;  
  
type  
    TWaveButton = class (TButton)  
    private  
        {Private declarations}  
        FSoundFile: string;  
    protected  
        {Protected declarations}  
    public  
        {Public declarations}  
        constructor Create (AOwner: TComponent); override;  
        procedure Click; override; //覆盖触发 OnClick 事件的 Click 虚方法  
    published  
        {Published properties and events}  
        property SoundFile: string read FSoundFile write FSoundFile;  
    end; {TWaveButton}  
  
procedure Register;  
  
implementation  
  
uses  
    mmSystem;
```

```

procedure TWaveButton.Click;
begin
    sndPlaySound (@ FSoundFile, snd_ Async or snd_ NoDefault );
    inherited Click;
end;

constructor TWaveButton.Create (AOwner: TComponent);
{Creates an object of type TWaveButton, and initializes properties.}
begin
    inherited Create (AOwner);
    FSoundFile := '* .wav';
end; {Create}

procedure Register;
begin
    RegisterComponents ('New', [TWaveButton]);
end; {Register}

end.

```

3. 处理 Windows 消息句柄

如果基类组件不提供触发事件的虚方法，你仍然可以通过直接处理 Windows 消息来获取组件对象状态改变信息。下面的代码显示了在 VCL 中 Windows 消息是怎样工作的：

```

TControl = class (TComponent)
private
    FOnClick: TNotifyEvent;
    ...
    procedure WMLButtonUp (var Message: TWMLButtonUp); message WM_ LBUTTONUP;
    ...
protected
    ...
    procedure Click; dynamic;
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
    ...
end; {TControl}

...

procedure TControl.Click;
begin
    if Assigned (FOnClick) then
        FOnClick (Self);
end;

procedure TControl.WMLButtonUp (var Message: TWMLButtonUp);
begin
    inherited;
    if csCaptureMouse in ControlStyle then
        MouseCapture := False;
    if csClicked in ControlState then
        begin
            Exclude (FControlState, csClicked);

```



```

    if PtInRect (ClientRect, SmallPointToPoint (Message.Pos)) then
        Click; {Call our virtual method that in turn calls event handler.}
    end;
    DoMouseUp (Message, mbLeft);
end;

```

显然，通过覆盖 Windows 消息句柄可以解决组件设计上的局限。

4. 应用模板方法设计模式

模板方法设计模式 (Template Method Pattern) 是体现继承强大功能的又一例。设计模式总结了面向对象编程中的一些有效解决方法。模板方法模式定义了一个算法的轮廓和骨架，而将算法的一些逻辑步骤的具体实现延迟到派生类中。使用模板方法模式的优点是，你可以快速简单地创建基类的不同新版本，并提供解决新问题的实现方法。具体地讲，你要先创建一个基类，提供一系列的算法步骤，作为模板方法。然后创建一个派生类，根据具体情况对需要改动的算法提供新的实现方法。尽管派生类可以改变某些算法的步骤，但算法步骤的实现顺序仍然由基类控制。示例程序 4-17 就是一个模板方法设计模式的例子。

示例程序 4-17 模板方法设计模式示例

```

unit InheritanceDemo;

interface

type
    TBasicDocumentEditorClass = class (TPersistent)
    protected
        function CurrentDocumentIsDirty: boolean; virtual;
        procedure SaveCurrentDocument; virtual;
        procedure OpenDocument (fileName: string); virtual;
        function GetNewDocumentNameToEdit: string; virtual;
    public
        procedure OpenNewDocument;
    end;

    TAdvancedDocumentEditorClass = class (TBasicDocumentEditorClass)
    protected
        procedure SaveCurrentDocument; override;
        procedure OpenDocument (fileName: string); override;
    end;

implementation

...

procedure TBasicDocumentEditorClass.OpenNewDocument; {public}
var
    documentName: string;
begin
    {这里列出了算法的步骤:}
    if CurrentDocumentIsDirty then
        SaveCurrentDocument;
    documentName := GetNewDocumentNameToEdit;

```

```

    OpenDocument (documentName);
end; |OpenNewDocument |

...

procedure TAdvancedDocumentEditorClass.SaveCurrentDocument;
begin
    ... {这里是新增功能 |
    inherited SaveCurrentDocument;
end; |SaveCurrentDocument |

procedure TAdvancedDocumentEditorClass.OpenDocument (fileName: string);
begin
    inherited OpenDocument (fileName);
    ... {这里是新增功能 |
end; |OpenDocument |

end.

```

总之，继承是创建组件、实现重用的重要途径。但使用继承来创建组件所能达到的效果往往取决于开发者对基类组件的熟悉程度以及该类组件针对重用的设计水准。所以设计者必须充分进行面向对象的思考，确保组件的可重用性。

4.5.3 合成与嵌入

1. 合成

通过合成，我们可以创建一个组件对象组，该对象组以新组件的形式出现，方便了我们的使用。合成多个可视化组件需要有一个容器组件，比如 TPanel、TCustomPanel 或 TWinControl。这些窗体组件是作为子组件的容器的。使用合成技术，你可以只公布子组件的必要属性和方法，如果不希望组件使用者介入，也可以设计时私下响应子组件的事件。合成的组件可以使组件的组合变得简单，并有利于管理好内部子组件之间的关系。

示例程序 4-18 就是一个简单的合成组件例子，该合成组件组合了两个按钮子组件（OK 和 Cancel 按钮对），公布了各个子组件的 OnClick 事件和 Caption 属性。

示例程序 4-18 组合了两个按钮子组件的合成组件 TOKCancelButton

```

unit OKCancel; {TOKCancelButton 合成组件}

interface

uses
    Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, Buttons;

type
    TOKCancelButton = class (TWinControl)
    {声明子组件}
        OKBtn: TBitBtn;
        CancelBtn: TBitBtn;
    private

```

```

    {用于公布事件的数据成员}
    FOnClick_OKBtn: TNotifyEvent;
    FOnClick_CancelBtn: TNotifyEvent;
    {用于公布属性的读写方法}
    procedure SetCaption_OKBtn (newValue: TCaption);
    function GetCaption_OKBtn: TCaption;
    procedure SetCaption_CancelBtn (newValue: TCaption);
    function GetCaption_CancelBtn: TCaption;
protected
    procedure Click_OKBtnTransfer (Sender: TObject); virtual;
    procedure Click_CancelBtnTransfer (Sender: TObject); virtual;
public
    constructor Create (AOwner: TComponent); override;
published
    {公布子组件的属性}
    property Caption_OKBtn: TCaption read GetCaption_OKBtn
        write SetCaption_OKBtn;
    property Caption_CancelBtn: TCaption read GetCaption_CancelBtn
        write SetCaption_CancelBtn;

    {公布子组件的事件}
    property OnClick_OKBtn: TNotifyEvent read FOnClick_OKBtn
        write FOnClick_OKBtn;
    property OnClick_CancelBtn: TNotifyEvent read FOnClick_CancelBtn
        write FOnClick_CancelBtn;

end;

procedure Register;

implementation

procedure TOKCancelButton.SetCaption_OKBtn (newValue: TCaption);
{为 OKBtn 子组件的 Caption 属性设置新值}
begin
    OKBtn.Caption := newValue;
end;

function TOKCancelButton.GetCaption_OKBtn: TCaption;
{从 OKBtn 子组件返回 Caption 属性值}
begin
    GetCaption_OKBtn := OKBtn.Caption;
end;

procedure TOKCancelButton.SetCaption_CancelBtn (newValue: TCaption);
begin
    CancelBtn.Caption := newValue;
end;

function TOKCancelButton.GetCaption_CancelBtn: TCaption;
begin
    GetCaption_CancelBtn := CancelBtn.Caption;
end;

```

|下一步,为每个子组件实现 Click 事件传递方法。这些方法检查是否分配了事件句柄 (双击 Object Inspector 中的事件可以分配事件句柄) |

```

procedure TOKCancelButton.Click_OKBtnTransfer (Sender: TObject);
|将 OKBtn 的 OnClick 事件传递到外部|
begin
    if assigned (FOnClick_OKBtn) then
        FOnClick_OKBtn (Self); |用 Self 取代子组件的 Sender|
end;

procedure TOKCancelButton.Click_CancelBtnTransfer (Sender: TObject);
begin
    if assigned (FOnClick_CancelBtn) then
        FOnClick_CancelBtn (Self);
end;

|最后设置位置、大小等参数,创建该组件及其子组件。还要直接为各个子组件动态分配事件传递方法。|

constructor TOKCancelButton.Create (AOwner: TComponent);
|创建 TOKCancelButton 类型的对象,初始化属性值。|
begin
    inherited Create (AOwner);
    Width := 75;
    Height := 56;
    OKBtn := TBitBtn.Create (Self);
    with OKBtn do
        begin
            Parent := Self;
            Kind := bkOK;
            SetBounds (0, 0, 75, 25);
            TabOrder := 0;
            OnClick := Click_OKBtnTransfer; |动态分配事件句柄|
        end; |OKBtn|

    CancelBtn := TBitBtn.Create (Self);
    with CancelBtn do
        begin
            Parent := Self;
            Kind := bkCancel;
            SetBounds (0, 31, 75, 25);
            TabOrder := 1;
            OnClick := Click_CancelBtnTransfer; |动态分配事件句柄|
        end; |CancelBtn|
    end;

    procedure Register;
    begin
        RegisterComponents ('New', [TOKCancelButton]);
    end; |Register|

end.

```

合成组件使我们能够在设计期就将可视化组件组合在一起,新的合成组件封装了各个子组件的复杂性及其内部关系,使得程序员(组件的使用者)在编程中更加容易使用。

2. 嵌入

如果说合成针对的是可视化组件对象,那么嵌入则针对的是非可视化组件对象了。嵌入组

件的步骤和创建合成组件类似。不同之处在于嵌入组件不需要容器。既可以在现存的类中嵌入组件，也可以派生新类并嵌入组件。

示例程序 4-19 是一个通过把 TTimer 嵌入到 TLabel 中而新创建的 TClockLabel 组件。

示例程序 4-19 把 TTimer 嵌入到 TLabel 中的 TClockLabel 组件

```

unit ClockLbl; {TClockLabel 组件}

interface

uses
  Windows, SysUtils, Messages, Classes, Graphics, Controls, Forms, Dialogs, Menus, StdCtrls,
  ExtCtrls;

type
  TClockLabel = class (TLabel)
  private
    {Private declarations }
    EmbeddedTimer: TTimer;
    {读写嵌入的 TTime 组件}
    procedure SetEnabled _ EmbeddedTimer (newValue: Boolean);
    function GetEnabled _ EmbeddedTimer: Boolean;
    procedure SetInterval _ EmbeddedTimer (newValue: Cardinal);
    function GetInterval _ EmbeddedTimer: Cardinal;
  protected
    {Protected declarations }
    procedure Timer _ EmbeddedTimerHandler (Sender: TObject); virtual;
  public
    {Public declarations }
    constructor Create (AOwner: TComponent); override;
  published
    {Published properties and events }
    {公布嵌入组件的属性}
    property Enabled _ EmbeddedTimer: Boolean read GetEnabled _ EmbeddedTimer write SetEnabled _
      EmbeddedTimer;
    property Interval _ EmbeddedTimer: Cardinal read GetInterval _ EmbeddedTimer write SetInterval _
      EmbeddedTimer;
  end;

  procedure Register;

implementation

{继承下来的组件读写方法}
procedure TClockLabel.SetEnabled _ EmbeddedTimer (newValue: Boolean);
begin
  EmbeddedTimer.Enabled := newValue;
end;

function TClockLabel.GetEnabled _ EmbeddedTimer: Boolean;
begin
  GetEnabled _ EmbeddedTimer := EmbeddedTimer.Enabled;
end;

```

```

procedure TClockLabel.SetInterval _ EmbeddedTimer (newValue: Cardinal);
begin
    EmbeddedTimer.Interval := newValue;
end;

function TClockLabel.GetInterval _ EmbeddedTimer: Cardinal;
begin
    GetInterval _ EmbeddedTimer := EmbeddedTimer.Interval;
end;

procedure TClockLabel.Timer _ EmbeddedTimerHandler (Sender: TObject);
begin
    Caption := TimeToStr (Time);
end;

constructor TClockLabel.Create (AOwner: TComponent);
|创建 TClockLabel 对象, 初始化属性值|
begin
    inherited Create (AOwner);
    |创建嵌入的对象变量|
    EmbeddedTimer := TTimer.Create (Self);
    EmbeddedTimer.OnTimer := Timer _ EmbeddedTimerHandler;
end;

procedure Register;
begin
    RegisterComponents ('CDK', [TClockLabel]);
end; {Register}

end.

```

对象的合成与嵌入是创建组件的有效方式, 虽然该功能很强大, 但没有对象的链接来得灵活。使用嵌入的好处是在设计应用程序的时候不必考虑链接其他的对象, 缺点是嵌入对象和属主对象的关系是紧耦合。链接克服了这个缺点。

4.5.4 链接

比如 Delphi 中新增的 TLabelledEdit 组件就是组合了 Label 和 Edit 组件的一个例子。

组件链接使我们能够以比较灵活的方式来合成新组件, 这样可以使应用程序开发者有更多的自由来控制链接的对象。在 VCL 中包括了很多组件链接的例子, 比如: TLabel 组件的 FocusControl 属性, 数据感知控件 (data-aware control) 的 DataSource 属性都是用于组件链接的。

1. 链接属性

链接对象的第一步是创建用于链接对象的属性, 并为其指定读写方法, 属性的类型要和链接对象的类型一致。例如:

```

TRTFFormatPanel = class (TCompoundComponentPanel)
...
private
    FRichEditControl: TRichEdit;
...

```

```
procedure SetRichEditControl (newValue: TRichEdit );
published
property RichEditControl: TRichEdit read FRichEditControl
                                write SetRichEditControl;
end; {TRTFFormatPanel }
```

2. 覆盖 Notification 方法

第二步是覆盖 Notification 方法。当设计期从 form 中加入或移除新组件时，或运行期销毁它时，将会调用 Notification 方法，所以通过覆盖 Notification 方法可以使我们的代码无需在这些时候访问链接的组件。覆盖 Notification 方法的声明类似下面这样：

```
TRTFFormatPanel = class (TCompoundComponentPanel)
...
protected
    procedure Notification (someComponent: TComponent; operation: TOperation); override;
...
end; {TRTFFormatPanel }
```

然后完成实现代码：

```
procedure TRTFFormatPanel.Notification ( someComponent: TComponent ; operation:
TOperation );
begin
    inherited Notification (someComponent, operation);
    if operation = opRemove then {someComponent is about to be deleted.}
    begin
        if someComponent = FRichEditControl then
            FRichEditControl := nil;
        end;
    end; {Notification }
```

当你在 form 中撰写代码访问链接的组件时，通常要检查并确认该组件是否存在：

```
if assigned (FRichEditControl) then
    FRichEditControl.Update;
```

Set 方法的实现代码可能要完成组件的初始化设置，并调用 FreeNotification 方法：

```
procedure TRTFFormatPanel.SetRichEditControl (newValue: TRichEdit );
begin
    if FRichEditControl < > newValue then
    begin
        FRichEditControl := newValue;
        if FRichEditControl < > nil then
            FRichEditControl.FreeNotification (Self);
        end;
    end;
```

注意 FreeNotification 是在 Delphi 2 以上的版本安装组件所必需的，它可以使得 Notification 方法跨窗体工作（指当链接组件和属主组件不在同一个窗体上的情况）。关于 FreeNotification 的详细介绍参见 9.4.4 节。

3. 自动挂接

正如前面所述，当从 form 中加入或移除组件时调用了 Notification 方法，我们可以考虑利用

该信息来自动挂接匹配的组件。以下代码实现了目标组件和属主组件的自动挂接，但要保证当链接目标组件时，属主组件必须已经在 form 中：

```
procedure TRTFFormatPanel.Notification (AComponent: TComponent; operation: TOperation);
begin
  inherited Notification (AComponent, operation);
  if operation = opInsert then {AComponent 加入时}
  begin
    if (AComponent is TRichEdit) and not assigned (FRichEditControl) then
      FRichEditControl := AComponent; {关联引用}
    end;
  if operation = opRemove then {AComponent 移除时}
  begin
    if AComponent = FRichEditControl then
      FRichEditControl := nil;
    end;
  end; {Notification }
```

组件链接以一种强大而灵活的方式在应用程序设计期组合对象。该技术可以让应用程序开发人员将指定类型的组件链接到自己开发的组件中。链接组件的类型也可以是指定类型的派生类，这对消除组件和组件之间，组件和类型之间的强耦合很有帮助。

第5章 深入多态

5.1 认识多态

在英文中，poly 是多的意思，morph 是形态的意思，多态（polymorphism）意味着可以具有多种形式或形态。在 OOP 中，它与类的继承密切相关。实际上，是继承产生了多态。

继承和多态都是面向对象编程中的重要概念，这个概念既解决了对客观世界复杂性的抽象，又可以高效地实现软件的重用。

据说上帝在创造人类的时候，也只造了亚当和夏娃，而且是用面向对象的方法。通过继承和多态，由亚当和夏娃的子孙繁衍出犹太人、亚述人、腓尼基人等不同的种族，他们有自己的姓氏、肤色、语言等不同的特征。上帝在实现人类多样化的进程中并没有花太多的精力，因为他发明了一种叫做“基因”的对象，并由该对象自动完成了继承和多态的功能，我们通常称之为“基因遗传”和“基因突变”。

另一个大家熟知的例子是有关战国末年公孙龙“白马非马”的诡辩。说白马是马，是因为白马继承了马的特性，复用了马的功能；说白马非马，是因为白马除了继承马的普遍性之外，还增加了“白色”的特殊性，使其不是一般意义上的马（比如：比黑马更具有观赏性）。而不论是白马、黑马还是其他各色的马，都体现了马的多种形态，是马的多态。

如果从哲学上来看，继承就是对共性的继承，体现了事物的普遍性；多态是对特性的肯定，体现了事物的多样性。

在面向对象的编程中，多态更多地体现在对象具体行为上的差异。所以，许多看似相同的行为（因为名称相同），具体执行的过程却大相径庭。这种差异通过 OOP 实际上是将“是什么”（What）从“怎么做”（How）之中抽离出来，从而提供了接口和实现分离这样一个重要特性。这就可以让我们进一步改善程序的结构和性能，开发出可扩展的程序。

为了帮助读者理解多态，我们来看下面这样一个例子。

如果有一天你参加一个国际会议，可能会遇见许多来自世界各地的朋友。大家见了面，互相问候，说的都是“你好”，但各国朋友的发音却千差万别。同样都是问候，问候的具体形式却各不相同，这正是多态“一个接口，多种实现”的典型例子。关于这个“来自世界的问候”的例子后面我们还会反复用到，并给出具体实现。

在 Delphi 中一旦一个类继承了另一个类，这个类就是另一个类的派生类。派生类继承了基类所有的字段、方法和属性。Delphi 仅支持单亲继承，故一个类只有一个基类，该基类也可以有它自己的基类。从而一个类继承了每一个祖先类的字段、属性和方法。这种继承可以是无条件地全部继承旧方法，也可以是重新编写方法覆盖原来的旧方法。这样就意味着，在派生类中可以通过覆盖机制来增强或改变对象的行为，从而实现多态。

在 Delphi 的类声明中，通过将基类中要覆盖的方法事先声明为虚（virtual）方法或动态（dynamic）方法，就可以在派生类中实现对该方法的覆盖。同时，采用多态的编程技巧可以很

容易地实现一个接口，多种实现，增加代码的可复用性和可维护性。

5.2 重载与覆盖

在 Delphi 中，重载 (overload) 和覆盖 (override) 是经常容易混淆的概念。特别是在许多 Delphi 书中，这两个术语使用非常混乱。其中，有些书中把 “overload” 和 “override” 分别译为 “过载” 和 “重载”，我认为这样翻译也不好，因为第一，重载和覆盖是 OOP 中的重要术语，应该和 C++、C#、Java 等面向对象语言中的译法统一；第二，overload 和 override 的翻译应该体现其概念本身的含义，而不要拘泥于字面，否则容易造成望文生义，引起误解。

下面我们来仔细体会一下重载和覆盖的不同含义。

5.2.1 重载

我们知道编译器在编译程序时会给每一个过程分配一个地址，因此过程名要求惟一。这也是我们在编程时养成的良好习惯，即在程序中自觉地为每一个过程准备一个惟一的名称。

现在我们假设有这样一个名为 queryEmployee() 的过程，需要从数据库中按照一定的条件查询员工信息。如果在设计时用户需求简单，只需要按照工号查询，则该过程只需要定义一个参数，即 queryEmployee (parameter: Integer) 就可以了。如果用户需要按照工号、按照姓名、按照生日等多种方式查询，我们就无法使用 queryEmployee (paramete: Integer) 了。因为我们使用的参数类型有 Integer、String、Date 三种。通常的解决办法就是使用三种不同名称的过程来分别实现不同条件的查询，它们是：

```
queryEmployeeById (paramete: Integer);  
queryEmployeeByName (paramete: String);  
queryEmployeeByBirthday (paramete: Date);
```

这样写虽然可解决眼前的问题，但是过程使用了不同的名称，使得原来含义清晰的 queryEmployee (查询员工) 名称变得冗长费解，而且不易扩展。试想，如果用户又要新增按照工号、姓名和生日的组合查询，是不是还要使用名称更复杂的新过程：

```
queryEmployeeByIdAndNameAndBirthday (Id: Integer; Name: String; Birthday: Date);
```

如果我们采用新的重载过程的方法，就可以通过 Overload 限定符实现同名过程的不同实现了。下面我们来重写以上不同条件的查询：

```
queryEmployee (Id: Integer); Overload;  
queryEmployee (Name: String); Overload;  
queryEmployee (Birthday: Date); Overload;  
queryEmployee (Id: Integer; Name: String; Birthday: Date); Overload;
```

由此可见，名字相同而参数的数据类型和数量不同的过程，称之为重载过程。这是因为编译器在编译时不仅使用了过程名称的信息还使用了参数类型和数量的信息，通过组合这些信息可以确定分配地址时需要的惟一标识，通常称之为签名 (signature)。现在对于所有同名的 queryEmployee 过程的引用，编译器都可以基于参数的数据类型进行正确的解析。

于是程序员只要记住 queryEmployee 过程，无论是使用 queryEmployee (25) 还是 queryEmployee

(“张三”)都会解析到正确的实现。由于 queryEmployee 过程可能已经在其他地方引用,当用户需求变化要求新增不同的查询条件时,重载新的 queryEmployee 过程并不会影响到原来的 queryEmployee 过程,更不会应为不断增多的新名称导致程序员无所适从。所以重载增加了编程的灵活性,使得程序更易于扩展,并保持了简洁优美的风格。

实际上重载不限于类的成员过程(即不限于对方法的重载),重载适用于所有过程。这就是说,重载不同于覆盖,它不是面向对象专有的。

在 Delphi 自身的一些系统单元中,我们可以找到大量的重载函数,对我们提高编程技巧有着很好的示范作用。比如主要用于计量单位间的数值转换的 ConvUtils 单元定义了一些与单位转换工具相关的过程和函数,该单元定义的函数中,有几个函数是单元的核心,其中的 Convert 函数是单位的转换函数,是编程最常用的函数,它被声明为两个重载函数,即:

```
function Convert (const AValue: Double; const AFrom, ATo: TConvType): Double; overload;  
function Convert (const AValue: Double; const AFrom1, AFrom2, ATol, ATo2: TConvType): Double;  
overload;
```

在第一个重载函数中, AValue 参数传入当前要转换的数值,其单位由 AFrom 参数传入,要转换的目标单位由 ATo 参数传入,函数返回转换的结果。TConvType 被声明为 Word 类型。在编程时,可以用类似如下代码实现常见单位间的转换:

```
Convert (31, tuCelsius, tuFahrenheit); //31 摄氏度转换为华氏温度。  
Convert (80, tuSeconds, tuMinutes); //将 80 秒转为分钟。  
Convert (40, duDecimeters, duMeters); //将 40 分米转为米。
```

第二个重载函数主要用于复杂的单位转换,比如将以英里/小时为单位的速度转换为以米/分钟为单位的速度。AFrom1/AFrom2 是要转换的数值的单位,而 ATol/ATo2 是转换后的目标单位。例如:

```
Convert (10, duMiles, vuGallons, duKilometers, vuLiter);  
//将汽车耗油量 10 英里/加仑转换为公里/升。
```

5.2.2 覆盖

当一个类被继承时,它的方法也就被其派生类继承。这种继承可以是无条件地全部继承旧方法,也可以是重新编写方法覆盖原来的旧方法。覆盖的本质就是以新方法替代同名的旧方法。这样就意味着,在派生类中可以通过覆盖机制来增强或改变对象的行为,从而实现多态。

对于基类要覆盖的方法,必须将其事先声明为虚方法,否则使用覆盖方法时会出错。虚方法有 virtual 和 dynamic 两种限定符声明方式。这和 Delphi 的内部机制有关,后面我会详细介绍。

为了使读者有比较深刻的体会,下面我们就用“来自世界的问候”这个例子演示如何让不同国家的人说问候语“你好”。

为了将逻辑和界面分离,我们先创建了一个名为 uSayHello 的单元文件(参见示例程序 5-1),用于存放类。这里有一个名为 TMan 的基类,以及从 Man 类继承的派生类 TChinese、TAmerican、TFrench 和 Tkorean。它们可以实现一个 SayHello 方法,方法是类的行为,它既可以是过程,也可以是函数,我们的 SayHello 方法是一个函数,返回一个图片文件名,该图片上面

印有问候语。

注意 如果虚方法是过程，则称该过程为虚过程；如果是函数，则称该函数为虚函数。

示例程序 5-1 uSayHello

```
unit uSayHello;

interface

type
  TMan = class (TObject)
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; virtual;
    function SayHello: string; virtual;
  end;

  TChinese = class (TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TAmerican = class (TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TFrench = class (TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TKorean = class (TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

implementation

constructor TMan.create;
begin
  Name: = '张三';
  Language: = '中文';
  SkinColor: = '黄色';
end;
```

```
constructor TChinese.create;
begin
    inherited;
end;

constructor TAmerican.create;
begin
    Name: = 'Lee';
    Language: = '英文';
    SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
    Name: = '苏菲';
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;

function TMan.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

end.
```

大家注意到 SayHello 方法在基类 TMan 中是一个虚方法，以利于实现覆盖。派生类 TChinese、

TAmerican、TFrench 和 TKorean 的 SayHello 方法分别使用 override 限定符来实现覆盖。这样，不同国家的人就可以用本国语言来“SayHello”了。

接下来我们设计一个界面，如图 5-1 所示，用来创建中国人、美国人、法国人和韩国人等不同对象，并显示他们的名字、语言、肤色以及问候语。该界面文件名是 ufrmSayHello，代码见示例程序 5-2。



图 5-1 “来自世界的问候”窗体设计界面

示例程序 5-2 ufrmSayHello

```
unit ufrmSayHello;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TfrmSayHello = class (TForm)
    GroupBox1: TGroupBox;
    edtName: TLabelledEdit;
    edtSkinColor: TLabelledEdit;
    edtLanguage: TLabelledEdit;
    btnUSA: TButton;
    btnKorean: TButton;
    btnCN: TButton;
    btnFrench: TButton;
    Image1: TImage;
    procedure btnUSAClick (Sender: TObject);
    procedure btnCNClick (Sender: TObject);
  end;

end.
```



```
    procedure btnFrenchClick (Sender: TObject);
    procedure btnKoreanClick (Sender: TObject);
private
    {Private declarations}
public
    {Public declarations}
end;

var
    frmSayHello: TfrmSayHello;

implementation

uses uSayHello;

{$R *.dfm}

procedure TfrmSayHello.btnUSAClick (Sender: TObject);
var
    AMan: TMan;
begin
    AMan := TAmerican.create;
    edtName.Text := AMan.Name;
    edtLanguage.Text := AMan.Language;
    edtSkinColor.Text := AMan.SkinColor;
    image1.Picture.LoadFromFile (AMan.sayHello);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject);
var
    AMan: TMan;
begin
    AMan := TChinese.create;
    edtName.Text := AMan.Name;
    edtLanguage.Text := AMan.Language;
    edtSkinColor.Text := AMan.SkinColor;
    image1.Picture.LoadFromFile (AMan.sayHello);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject);
var
    AMan: TMan;
begin
    AMan := TFrench.create;
    edtName.Text := AMan.Name;
    edtLanguage.Text := AMan.Language;
    edtSkinColor.Text := AMan.SkinColor;
    image1.Picture.LoadFromFile (AMan.sayHello);
end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject);
var
    AMan: TMan;
begin
```

```

AMan: - TKorean.create;
edtName.Text: = AMan.Name;
edtLanguage.Text: = AMan.Language;
edtSkinColor.Text: = AMan.SkinColor;
image1.Picture.LoadFromFile (AMan.sayHello);
end;

end.

```

我们运行这个程序，单击“韩国人”按钮，此时用 TKorean 类创建了一个名为“金知中”的韩国人实例，他用韩语向大家问候，如图 5-2 所示。

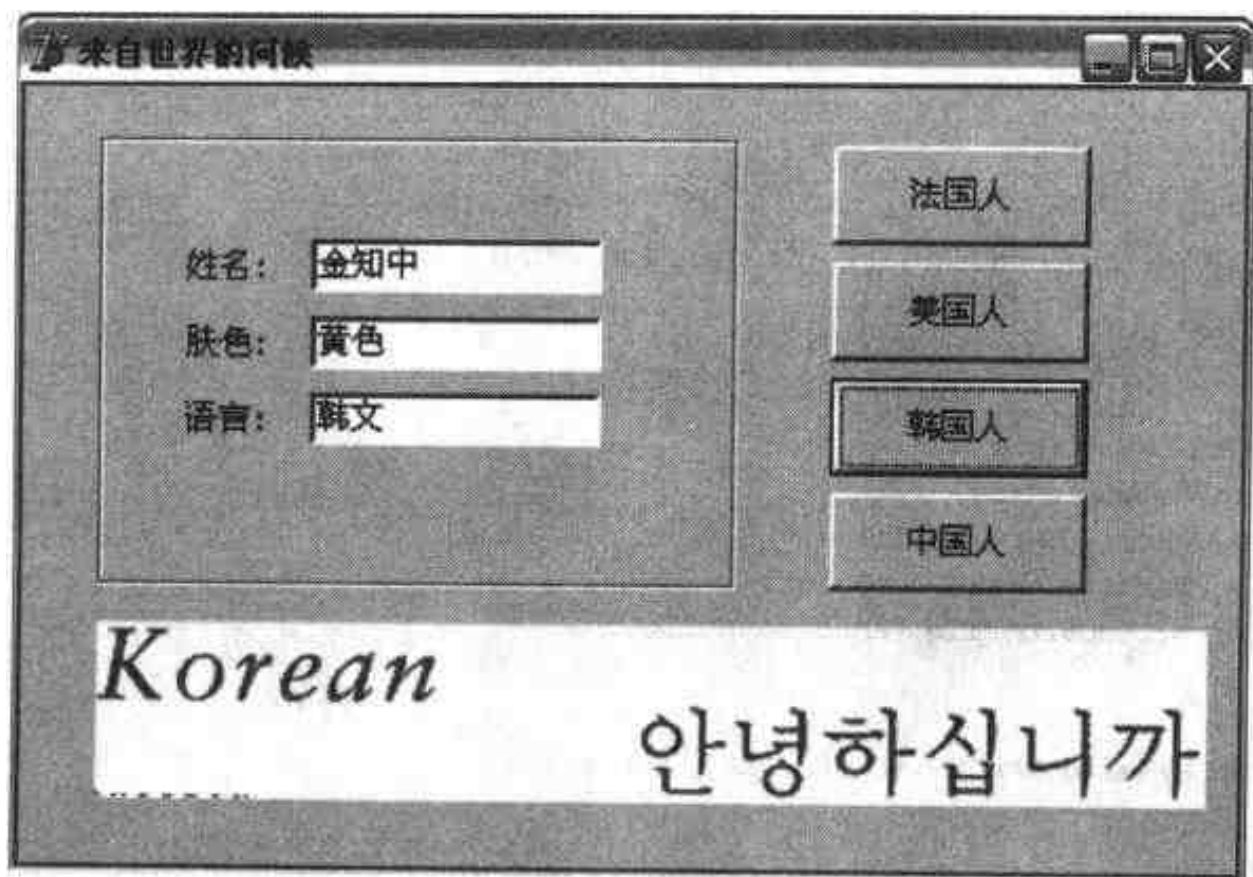


图 5-2 “来自世界的问候”窗体运行时演示了多态的作用

注意，这里 TKorean 类继承了 TMan 类，并覆盖了其中的 SayHello 方法。如果我们将示例程序 5-1 做以下改动：

```

type
  TMan = class (TObject)
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; virtual;
    function SayHello: string; virtual;
  end;
  .....
  TKorean = class (TMan)
  public
    constructor create; override;
    //function SayHello: string; override;注释掉该语句
  end;

```

```
implementation
.....
//function TKorean.SayHello; 注释掉该函数
//begin
//    Result := 'Korean.bmp';
//end;
```

重新运行程序，奇怪地发现“韩国人”用中文向大家问候。这是因为没有覆盖 SayHello 方法造成的，这样 TKorean 就继承了基类的 SayHello 方法。这也说明了覆盖可以改变基类的行为，实现多态。

有时我们的某些派生类不需要对基类的方法进行完全覆盖，或者完全不需要覆盖，这并不影响我们声明覆盖方法。我们可以在派生类的覆盖方法中使用 inherited 语句来解决这类问题。在示例程序 5-1 中，我们可以看到，TMan 类的构造函数是虚函数（虚方法），在这个虚函数中已经将其特性初始化为一个中国人。

```
constructor TMan.create;
begin
    Name := '张三';
    Language := '中文';
    SkinColor := '黄色';
end;
```

所以 TChinese 类的构造函数虽然是一个覆盖方法，但仍然能够通过 inherited 语句来全部继承 TMan.create 的代码。而其他的派生类则是完全覆盖了 TMan.create 构造函数。

```
constructor TChinese.create;
begin
    inherited;
end;
```

在实际应用中，我们除了在覆盖方法中继承基类的代码，还可以继续添加新代码来扩展基类的方法，这既不是完全的继承，也不是完全的覆盖。比如：

```
constructor TClass.create;
begin
    inherited;
    //在这里添加一些新代码,完成新增的任务
    .....
end;
```

如果基类的构造函数或析构函数是虚方法，则可以在派生类中覆盖构造函数或析构函数，但是对析构函数的覆盖比较特殊。如果覆盖了析构函数，则必须最后调用基类的析构函数，如下所示：

```
destructor TClass.destory;
begin
    //自己的代码写在前面
    .....
    inherited; //最后调用基类的析构函数
end;
```

如果先调用基类的析构函数，有可能销毁了仍需被派生类使用的对象，这非常危险。

5.3 虚方法与动态方法

虚方法使用 `virtual` 限定符，动态方法使用 `dynamic` 限定符，无论是 `virtual` 限定符还是 `dynamic` 限定符均表示在基类中声明了一个虚方法，其目的是为了派生类能够覆盖该方法，实现多态。这两个限定符的语义是相同的，惟一不同的是，它们的实现方法和调用方法，这涉及到 Delphi 的编译机制。

Delphi 的编译器会自动维护用于添加虚方法的虚方法表 (VMT) 和用于添加动态方法的动态方法表 (DMT)。

虚方法表 (VMT) 存放的是类及其基类声明的所有虚方法的指针。每一个类都具有一个惟一的 VMT，每一个类或其祖先类的虚方法在 VMT 中都有一个人口。无论一个类是否有自己定义的虚方法，只要它继承了祖先类的虚方法，它也会有自己的 VMT，列出它继承的所有虚方法。因为每一个类都有一个自己的 VMT，Delphi 就可以使用 VMT 来识别一个类，实际上，一个类引用就是指向类的 VMT 的指针，调用 `classtype` 方法则返回指向 VMT 的指针。

由于基类和派生类的 VMT 之间过强的关联性，因此派生类继承了祖先类的虚方法，也就是说派生类的 VMT 的前面一部分必须与基类相同。而当基类和派生类不在同一个 DLL 或 EXE 中的时候，这个要求是很难满足的。基类一旦改变，派生类如果不重新编译，就将导致错误。另外，如果类的层次很深，比如要建立一个有数百个类的大型框架，此时 VMT 将会因不断继承的虚方法而变得非常庞大，导致效率不高。

为了解决这个问题，Delphi 提供了一个 DMT 的机制。每一个类的 DMT 中只维护部分的动态方法，也就是说 DMT 仅列出了该类所声明的动态方法，它并不包括从祖先类那里继承来的方法。

如图 5-3 所示，DMT 使用了一种独特的编号和检索系统，当调用动态方法时，会在 DMT 的 `Indexes` 数组中查找该方法的代号，一旦找到匹配该方法的代号，则方法指针到相应的地址中调用该方法。如果方法代号没有找到，则会继续搜索其基类。

这样，动态方法就解决了前面的问题。基类和派生类的方法表之间没有关联性，可以自由改动而不会互相影响。不过，虽然 `dynamic` 方法解决了 `virtual` 方法的问题，但是也付出了不小的代价：时间效率和可读性，因此也决定了该方案的应用面不广。

`dynamic` 方法一般用于：

- 虚函数很少或几乎不需改写的情况。这样有助于减少 VMT 的大小。至于运行速度则没有什么提高，毕竟 VMT 的访问速度是常数级。
- 基类需要经常更新而派生类不方便同步更新，对效率要求又不高的情况。一般的应用程序都可以使用。
- 如果系统的类体系本身过于庞大，又需要在继承树根部的类里声明许多虚方法，则使用 `dynamic` 方法可以显著改善 VMT 的效率。比如：VCL 中的 `TControl` 就声明了许多 `dynamic` 方法。

通常我们可以将 `dynamic` 方法看成是另一种虚方法。实际上，大多数情况下使用 `virtual` 方

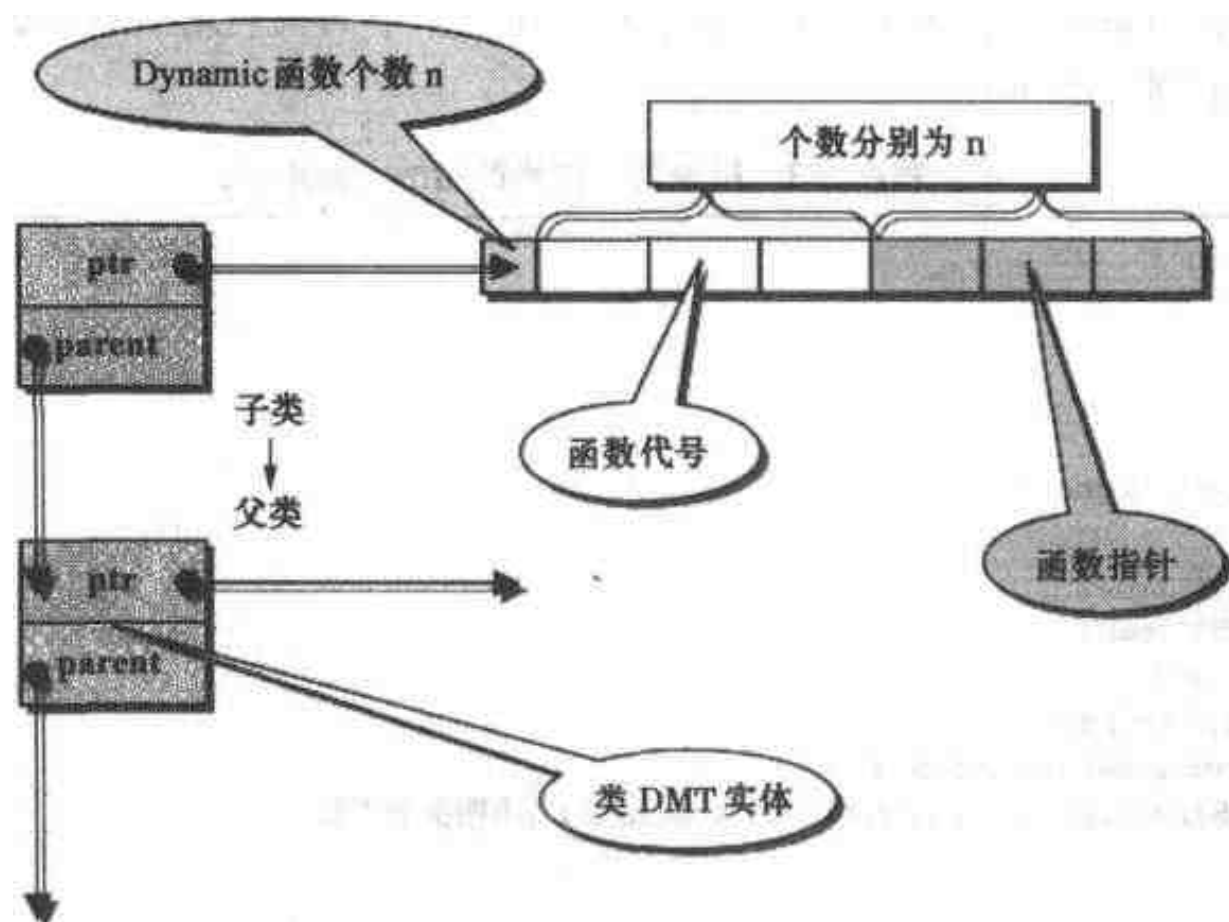


图 5-3 类的 DMT 图解

法要比 dynamic 方法快，占用内存少。

注意 声明方法时，virtual 和 dynamic 限定符必须位于 reintroduce 和 overload 限定符之后，abstract 限定符之前。

5.4 抽象类与抽象方法

常常遇到这样的情况：我们在祖先类内定义了一个方法，指望它的派生类能够继承，并且使用覆盖的方法具体化、实用化。但这个方法对于自身而言，没有必要编写任何代码，有时也实在不知道该编写什么代码。这时，我们把这个方法称为抽象方法，拥有抽象方法的类称为抽象类。抽象类给后代类提供一个公共的方法，但对于自身而言，没有任何意义。

换句话说，所谓抽象类就是至少有一个方法定义为抽象方法的类。抽象类是无需实例化的类，它提供的抽象方法为派生类定义了接口，它的任何派生类都必须实现这些方法。

抽象方法使用限定符 abstract 来声明，它是用来说明那些并没有实现的虚方法或动态方法的，这样编译器就可以在虚方法表（VMT）中保留一个位置或者分配一个动态方法代号。记住，抽象方法首先必须是虚方法或动态方法，abstract 限定符必须跟在 virtual、dynamic 或 override 之后。抽象方法定义如下：

```
procedure |或者 function| 方法名 (参数表) virtual abstract
```

或者

```
procedure |或者 function| 方法名 (参数表) dynamic abstract
```

抽象方法必须是一个虚方法或者动态方法，它与一般的虚方法和动态方法的不同之处在于抽象方法只对其进行定义，而不对其做任何实现，只在后代类中实现覆盖的地方进行实现。而

一般的虚方法或者动态方法必须在定义的类中也要实现。下面我们用抽象方法来修改示例程序 5-1uSayHello（见示例程序 5-3）：

示例程序 5-3 用抽象方法修改后的 uSayHello

```
unit uSayHello;

interface

type
  TMan = class(TObject)
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; virtual;
    function SayHello: string; virtual; abstract; //声明抽象方法
  end;

  TChinese = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TAmerican = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TFrench = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TKorean = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

implementation

constructor TMan.create;
begin
  Name: = '张三';
  Language: = '中文';
  SkinColor: = '黄色';
end;

constructor TChinese.create;
```

```
begin
    inherited;
end;

constructor TAmerican.create;
begin
    Name: = 'Lee';
    Language: = '英文';
    SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
    Name: = '苏菲';
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;
```

| = = = 声明了抽象方法后,这部分实现代码不再需要了。我已将这段代码注释掉了。 = = =

```
function TMan.SayHello;
begin
    Result: = 'chinese.bmp';
end;
{
```

//以下派生类(子类)通过覆盖实现了抽象方法

```
function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

end.
```


大家可以发现在“来自世界的问候”这个例子中，我们并没有在界面中（ufrmSayHello 单元）实例化 TMan 类，因为实际上并不存在一个抽象的人。只有具体的中国人、法国人才是我们需要的有意义的对象。因此，我们完全可以将 TMan 作为一个抽象类，并定义抽象方法 SayHello，作为一个公用接口使用。这样我们就无需考虑 TMan 的 SayHello 方法是如何实现的，并把这个工作留给派生类去具体实现。在示例程序 5-3 中，我注释掉了 TMan.SayHello 方法。由于我们遵从了界面单元（界面类）与实现逻辑单元（逻辑类）分离的编程思想，所以改动 uSayHello 并不影响到界面程序。这就是封装的好处。

如果不想在派生类中实现一个基类的抽象方法，就可以在派生类定义中忽略这个方法或者使用 override 和 abstract 限定符来声明这个方法。但是，如果程序员在派生类中忘记实现抽象方法或者用抽象类来创建实例，此时编译器将发出一个警告。

如果尝试为抽象类创建一个实例（构造一个对象），并调用它的抽象方法，Delphi 就会调用 AbstractErrorProc 过程或产生一个运行时错误 210（EAbstractError）。

多态依赖于抽象方法以及虚方法的概念，同时也和继承密切相关。因此我们往往定义一些底层的对象，然后将其中的某些实现定义为抽象的，也就是说我们仅仅定义了接口，而没有定义具体的实现细节。按照这样的思路，我们还会定义多个派生（继承）的对象，在这些对象中真正实现那些在祖先类中未曾实现的细节。这就使得我们先前定义的底层类，具有多态的特性。这种机制的好处在于，我们使用这些类的时候，只要一套代码，就可以完成多种功能。而惟一需要改变的就是创建对象的实例的那一部分。

5.5 类的类型转换

通过前面“来自世界的问候”的那个例子，我们已经初步体验到多态的作用。对于示例程序 5-1 可能会有读者（特别是习惯于面向过程的读者）不太明白下面代码的写法：

```
var
  AMan: TMan;
begin
  AMan := TKorean.create;
  .....
end;
```

大家最熟悉的写法是：

```
var
  AMan: TMan;
begin
  AMan := TMan.create;
  .....
end;
```

在实现部分的第一行中（即 begin 开始的下一行），建立了一个 TMan 类型的对象，并将其赋予 TMan 类型的变量 AMan，这是很正常的事。但是，建立了一个 TKorean 类型的对象，并将其赋予了 TMan 类型的变量 AMan，这看上去有点令人吃惊，不过这种写法是完全合法的。

众所周知，Delphi 是一种类型定义严格的语言，你不能将某个类型的值赋予不同类型的变

量，例如将一个整型值赋予布尔型变量，将会导致出错。但是，这个规则在涉及到 OOP 领域时，出现了一个重要的例外。那就是可以将一个派生类的值赋予一个基类类型的变量。但倒过来却是不行的，一个基类的值决不能赋予一个派生类类型的变量。这在 OOP 语言中也叫做“向上转型”（upcasting），即类类型向上转换，向基类转换。

5.5.1 向上转型

为什么类类型可以向上转型呢？是因为派生类可以继承基类的所有接口，包括方法和数据成员，也就是说派生类是基类的一个超集。在图 5-4 中，我们可以看到这种继承关系。如果，TKorean 类型的对象向上转型，它仍然可以在基类中找到 SayHello 接口，并通过后期绑定（late binding，也译为“晚绑定”）在运行时由 TKorean.SayHello 来实现该接口。

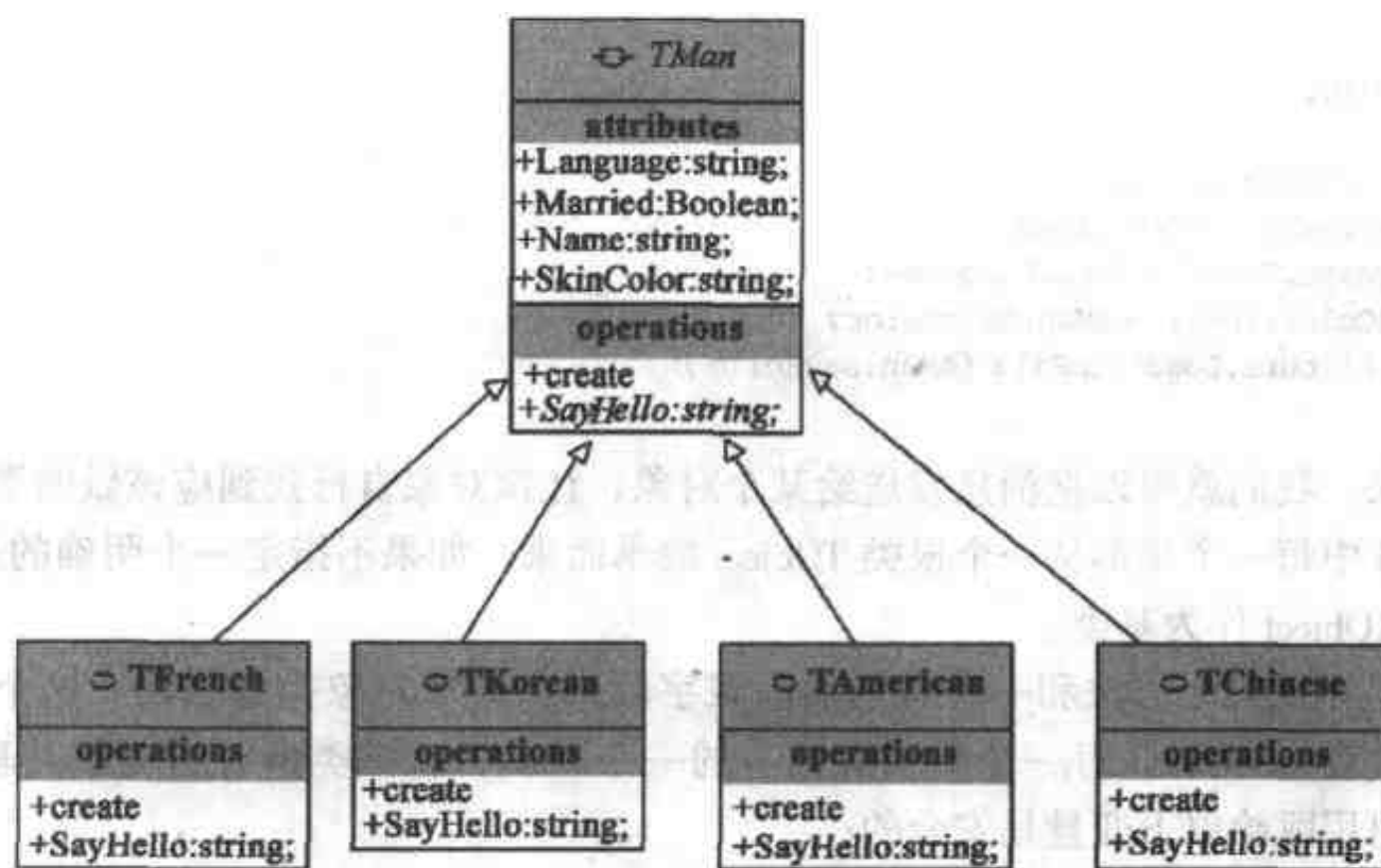


图 5-4 TMan 和其派生类的继承关系

注意 所谓绑定（binding）是指建立函数调用指针和函数具体实现之间的关联。如果绑定动作先于程序执行（由编译器事先编连好），则称为先期绑定或早绑定（early binding）；如果绑定动作在程序执行时确定（在运行时动态绑定），则称为后期绑定或晚绑定（late binding）。

那么，这种 OO 规则在编程中究竟有什么用处呢？

如果不用多态，我们可以为不同国家的人们分别写出不同的问候方法：

```
function SayHelloInEnglish: string;
function SayHelloInChinese: string;
function SayHelloInKorean: string;
.....
```

然后在其他程序中这样调用：

```
var
```

```
    AKorean: TKorean;  
begin  
    AKorean: = TKorean.create;  
    image1.Picture.LoadFromFile (AKorean.SayHelloInKorean);  
    .....  
end;
```

这样写程序显然要比示例程序 5-1 繁琐。试想，如果有一个人，他是 TMan 的一个实例，除非你在 TMan 中初始化他使用的默认语言，否则他是不知道自己会说什么话的。但是，如果他生长在韩国，他肯定会说韩语，生在法国，当然会说法语。所以，只要确定他是哪国人，TKorean 还是 TChinese，就能让他自己说自己的语言。这样的思维方式才是真正的面向对象思维方式，因为这样的对象有很强的自适应能力。所以才有示例程序 5-1 那样的写法：

```
procedure TfrmSayHello.btnKoreanClick (Sender: TObject);  
var  
    AMan: TMan;  
begin  
    AMan: = TKorean.create;  
    edtName.Text: = AMan.Name;  
    edtLanguage.Text: = AMan.Language;  
    edtSkinColor.Text: = AMan.SkinColor;  
    image1.Picture.LoadFromFile (AMan.sayHello);  
end;
```

这样一来，我们就可以把消息发送给某个对象，让该对象自行找到应该做的事了。

在 Delphi 中每一个类都从一个根类 TObject 继承而来。如果不指定一个明确的基类，Delphi 将自动使用 TObject 作为基类。

TObject 类声明几个方法和一个特别的隐藏字段来存放对对象类的引用。这个隐藏字段指向类的虚方法表 (VMT)。每一个类只有惟一的一个 VMT，且该类所有的对象共享类的 VMT。所以把对象引用赋给以下变量是安全的：

- 与该对象的类相同的类型变量。
- 该对象的祖先类类型变量。

换句话说，一个对象引用的声明类型并不一定与该对象的实际类型相同。反过来，将一个基类对象引用赋给一个派生类变量则是危险的，因为对象可能是不正确的类型。

要理解这一点其实并不难。大家还记得我们前面所举的一个“白马非马”的例子吗？白马是马，但是马不一定是白马。所以可以将一个“马”类型的引用赋给“白马”类型的变量，因为“白马”具有“马”的一切特征。但反过来，“马”不一定就具有“白马”的所有特征，“马”也可能是“黑马”，因此反向赋值是不确定的。

5.5.2 向下转型

我们可以发现，当使用向上转型时，实际上类类型的接口在缩小，因为基类不会比派生类拥有更多的接口，因此所有经由基类接口发送的消息都保证会被接受，也就是说向上转型是绝对安全的，惟一的问题是在派生类中扩充的接口在基类中无法使用。所以当使用向上转型后，便无法调用派生类中新增的方法了。

比如，在“来自世界的问候”那个例子中，如果浪漫的法国人问候时除了 SayHello，还有拥吻的动作，我们势必要新增 embrace 和 Kiss 方法，如图 5-5 所示。但是在示例程序 5-2 中我们使用了向上转型，所以无法调用 TFrench 类中新增的方法。实际上在 TMan 中根本就没有声明 embrace 和 Kiss 方法，TMan.embrace 和 TMan.Kiss 的写法无法通过编译。要解决这个问题，就必须使用向下转型（downcasting）。

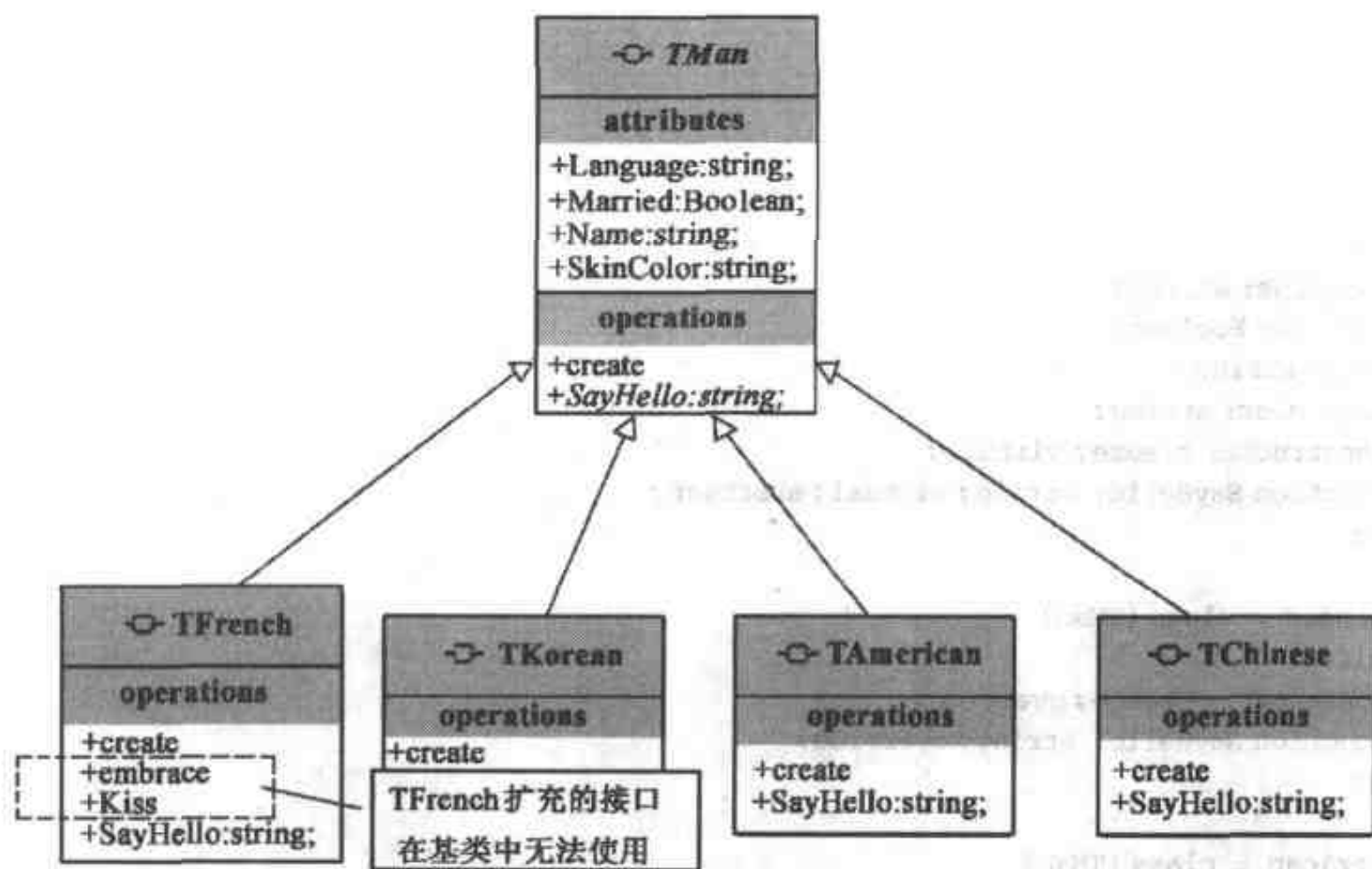


图 5-5 TMan 的派生类 TFrench 扩充了接口

向下转型与向上转型的概念正好相反，它是指将类类型向下转换，向派生类转换。正如前面所说，这种转换存在不确定因素，是危险的。为什么这样讲呢？因为向下转型扩大了类类型的接口，因此除非转型正确，否则难以保证经由派生类接口发送的消息都会被接受。

好在 Delphi 保留了 Pascal 强大的类型检查功能，因而编译器可以执行基于一个对象引用的声明类型的编译时检查。这样，所有方法都必须是声明过的类的一部分，且编译器执行对函数和过程参数的例行检查。编译器并不一定将方法调用绑定到一个指定的方法实现。如果该方法是虚的，Delphi 则等到运行时间，并使用对象的真实类型来决定调用哪一个方法实现。

为了保证转型正确，我们用 is 运算符来测试对象的真实类类型。如果类引用是对象的类或者它的任何祖先类，则返回真。如果对象引用是 nil 或者错误的类型，则返回假。例如：

```

//测试 AFrench 的类类型
if AFrench is TFrench then...
//如果 AFrench 不为 nil,则肯定判为真,因为所有类都是由 TObject 派生
if AFrench is TObject then...

```

也可以用类型转换来获得不同类型的对象引用。类型转换并不改变对象，它只是给你一个新的对象引用。通常，应该使用 as 运算符来进行类型转换。as 运算符自动检查对象的类型，并且如果对象的类不是目标类的派生类的话，将引发一个运行时错误。SysUtils 单元把运行时

错误映射到一个 EInvalidCast 异常中。

下面我们来看改动过的“来自世界的问候”程序。我们首先在 uSayHello 中为 TFrench 新增 embrace 和 Kiss 方法，如示例程序 5-4 所示。

示例程序 5-4 TFrench 新增方法后的 uSayHello

```
unit uSayHello;

interface

type
  TMan = class(TObject)
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; virtual;
    function SayHello: string; virtual; abstract;
  end;

  TChinese = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TAmerican = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

  TFrench = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
    function Kiss: PChar;
    function embrace: PChar;
  end;

  TKorean = class(TMan)
  public
    constructor create; override;
    function SayHello: string; override;
  end;

implementation

constructor TMan.create;
begin
  Name: = '张三';
  Language: = '中文';
```

```
    SkinColor: = '黄色';
end;

constructor TChinese.create;
begin
    inherited;
end;

constructor TAmerican.create;
begin
    Name: = 'Lee';
    Language: = '英文';
    SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
    Name: = '苏菲';
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;

{ == TFrench 新增方法,这部分在基类中无法使用 == }
function TFrench.Kiss;
begin
    Result: = '正在亲吻 ...';
end;

function TFrench.embrace;
begin
    Result: = '正在拥抱 ...';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;
```

```
function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

end.
```

接下来，在 `ufmSayHello` 单元中，我们先使用了向上转型，实现 `TMan` 类型 `SayHello` 的多态；然后使用了向下转型，将 `TMan` 类型的变量 `AMan` 转成 `TFrench` 类型变量 `AFrench`，这样就可以使用 `TFrench` 新增的 `embrace` 和 `Kiss` 方法了（见示例程序 5-5）。于是，点击法国人按钮，向下转型后出现了法国人浪漫的问候方式：亲吻加拥抱，如图 5-6 所示。

示例程序 5-5 用向下转型修改后的 `ufmSayHello`

```
unit ufmSayHello;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, ExtCtrls;

type
    TfrmSayHello = class (TForm)
        GroupBox1: TGroupBox;
        edtName: TLabelledEdit;
        edtSkinColor: TLabelledEdit;
        edtLanguage: TLabelledEdit;
        btnUSA: TButton;
        btnKorean: TButton;
        btnCN: TButton;
        btnFrench: TButton;
        Image1: TImage;
        procedure btnUSAClick (Sender: TObject);
        procedure btnCNClick (Sender: TObject);
        procedure btnFrenchClick (Sender: TObject);
        procedure btnKoreanClick (Sender: TObject);
    private
        {Private declarations}
    public
        {Public declarations}
    end;

var
    frmSayHello: TfrmSayHello;

implementation

uses uSayHello;

{$R *.dfm}
```



```
procedure TfrmSayHello.btnUSAClick (Sender: TObject );
var
  AMan: TMan;
begin
  AMan := TAmerican.create;
  edtName.Text := AMan.Name;
  edtLanguage.Text := AMan.Language;
  edtSkinColor.Text := AMan.SkinColor;
  image1.Picture.LoadFromFile (AMan.sayHello);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject );
var
  AMan: TMan;
begin
  AMan := TChinese.create;
  edtName.Text := AMan.Name;
  edtLanguage.Text := AMan.Language;
  edtSkinColor.Text := AMan.SkinColor;
  image1.Picture.LoadFromFile (AMan.sayHello);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject );
var
  AMan: TMan;
  AFrench: TFrench;
begin
  AMan := TFrench.create;
  edtName.Text := AMan.Name;
  edtLanguage.Text := AMan.Language;
  edtSkinColor.Text := AMan.SkinColor;
  image1.Picture.LoadFromFile (AMan.sayHello);
  //以下将使用向下转型实现派生类中新增的方法
  if AMan is TFrench then
  begin
    AFrench := AMan as TFrench;
    application.MessageBox (AFrench.kiss, '问候',
      MB_ICONINFORMATION + MB_OK);
    application.MessageBox (AFrench.embrace, '问候',
      MB_ICONINFORMATION + MB_OK);
  end
  else
  begin
    AFrench := nil;
    {//以下使用的是另一种向下转型方法,但不推荐使用。
    AFrench := TFrench (AMan);
    application.MessageBox (AFrench.kiss, '问候', MB_ICONINFORMATION + MB_OK);
    application.MessageBox (AFrench.embrace, '问候', MB_ICONINFORMATION + MB_OK);
  }
  end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject );
var
  AMan: TMan;
begin
```

```

AMan := TKorean.create;
edtName.Text := AMan.Name;
edtLanguage.Text := AMan.Language;
edtSkinColor.Text := AMan.SkinColor;
image1.Picture.LoadFromFile (AMan.sayHello);
end;

end.

```



图 5-6 向下转型后出现了法国人浪漫的问候方式

还有一种向下转型的方法是在常规对象转换中使用目标类的名称，类似于函数调用。这种风格的类型转换并不检查转换的有效性，故只有当你知道它是安全的时候才能使用它，但我不推荐使用。

//以下使用的是另一种向下转型方法,但不推荐使用。

```

AFrench := TFrench (AMan);
application.MessageBox (AFrench.kiss, '问候', MB_ICONINFORMATION + MB_OK);
application.MessageBox (AFrench.embrace, '问候', MB_ICONINFORMATION + MB_OK);

```

较好的方法是先用 `is` 进行类型判断，再用 `as` 进行显式的转型，这样可以降低向下转型潜在的风险，如示例程序 5-5 中所示。

不少 Delphi 书在讨论类型转换时都对向下转型避而不谈或反对读者使用，这就使得我们在编程时许多问题得不到解决。由于向上转型会遗失类型信息（比如 `TFrench` 向上转型为 `TMan` 后就遗失了 `embrace` 和 `Kiss` 方法），而向下转型可以帮我们取回类型信息，所以掌握向下转型的使用方法将会非常有用。

5.6 多态和面向对象编程

多态提供了接口与实现的分离，增加了程序的可读性和可扩展性，这是面向对象编程的重

要优势。接口关心的是“是什么”，实现关心的是“怎么做”，多态除了把接口和实现分离（面向过程编程中也强调将接口和实现分离）还能够去除类型之间的耦合，这就更妙了！

从前面的例子中我们看到，继承机制不但可以将某个对象以其本身的类型去看待，也可以以其基类类型去看待。这种能力极为重要，有了这种能力，我们就可以将多个类型视为一个类型，写一份代码，同时作用于这些不同的类型。多态的方法调用模式使得派生于同一基类型的那些相似的类型有能力表现它们的不同行为。

在“来自世界的问候”的这个例子中，不论你是韩国人（TKorean）还是中国人（TChinese）都是 TMan 类型的变量，而且都调用了 sayHello 方法。但是，执行的结果是完全不同的，前者执行的是 TKorean.sayHello 的代码，而后者执行的是 TChinese.sayHello 的代码！其原因很简单，因为有的 AMan 被赋给了 TKorean 类型的对象，而有的 AMan 被赋给了 TChinese 类型的对象。也就是说，一个 TMan 类型的变量，当它调用 sayHello 方法时，所执行的代码是不确定的，可能执行 TKorean.sayHello 的代码，也可能执行的是 TChinese.sayHello 的代码，这取决于它运行时引用的是一个什么样的对象，这也是 sayHello 方法晚绑定的好处。

更重要的是在示例程序 5-2 的所有 Button Click 事件中，除了 AMan := TKorean.create；这行代码是随着不同国家的人类型变化外，其他代码都是完全一样的。

AMan.sayHello 这行代码不需要变化，程序可以根据不同的情况赋予该变量不同的对象，从而使它执行不同的代码。这就是多态的定义。现在我们把示例程序 5-2 稍做修改，如示例程序 5-6 所示。

示例程序 5-6 优化过的 ufrmSayHello 增强了代码的可复用性

```
unit ufrmSayHello;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, uSayHello;

type
  TfrmSayHello = class (TForm)
    GroupBox1: TGroupBox;
    edtName: TLabelledEdit;
    edtSkinColor: TLabelledEdit;
    edtLanguage: TLabelledEdit;
    btnUSA: TButton;
    btnKorean: TButton;
    btnCN: TButton;
    btnFrench: TButton;
    Image1: TImage;
    procedure btnUSAClick (Sender: TObject);
    procedure btnCNClick (Sender: TObject);
    procedure btnFrenchClick (Sender: TObject);
    procedure btnKoreanClick (Sender: TObject);
  private
    procedure sayhello (AMan: TMan);
```

```

public
  {Public declarations }
end;

var
  frmSayHello: TfrmSayHello;

implementation

{$R *.dfm}
//可以复用的代码
procedure TfrmSayHello.sayhello (AMan: TMan);
begin
  edtName.Text: = AMan.Name;
  edtLanguage.Text: = AMan.Language;
  edtSkinColor.Text: = AMan.SkinColor;
  image1.Picture.LoadFromFile (AMan.sayHello);
end;

//原先的事件代码变得异常简洁
procedure TfrmSayHello.btnUSAClick (Sender: TObject);
begin
  sayhello (TAmerican.create);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject);
begin
  sayhello (TChinese.create);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject);
begin
  sayhello (TFrench.create);
end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject);
begin
  sayhello (TKorean.create);
end;

end.

```

读者会惊讶地发现示例程序 5-6 利用多态这个非常重要的特点大大地增强了代码的可复用性。如前所述，只需要在按钮事件中简单地写下一行代码，就可以让程序执行不同的功能，因为虚方法同 TMan 的任何派生类都是兼容的，甚至连那些还没有编写出来的类也是一样；而程序员并不需要了解这些派生类的细节。

利用多态性写出来的代码，除了简洁还有维护性好的特点。我们以后只要在示例程序 5-6 中的 TfrmSayHello.sayhello (AMan: TMan) 中维护代码就可以了。

现在我们再回顾一下 uSayHello 单元（示例程序 5-3），注意到 TMan 的 sayhello 被修改成了

抽象方法，所以 TMan 是无需实例化的抽象类。抽象方法本身不能够做任何事情，必须在派生类中被覆盖并实现，才能够完成有意义的工作。但抽象方法的存在，相当于为基类留下了一个接口，当程序将一个派生类的对象赋予基类的变量时，基类的变量就可以调用这个方法，当然此时它运行的是相应的派生类中覆盖该方法的代码。如果没有这个抽象方法，基类的变量就不能调用它，因为它不能调用一个只在派生类中存在而在基类中不存在的方法！

派生类在重新定义了基类的抽象方法后，由于重新定义的派生类的方法地址无法给出，其调用地址在编译期间无法确定，故基类指针必须根据覆盖它的不同的派生类指针在运行期动态地调用属于派生类的抽象方法。这样调用地址的方法显然是晚绑定。Bruce Eckel 在《Thinking in Java》中说，“Don't be fooled: If it isn't late binding, it isn't polymorphism”（不要犯傻，如果它不是晚绑定，它就不是多态）。实际上，我们之所以要用抽象方法（注意，它本身就是虚方法或动态方法）来实现晚绑定，是因为我们有时候根本无法确定该方法如何实现（可能有多种实现），所以留下的仅仅是一个接口而已。但是这个接口可以通过不同的派生类来实现，这就增加了程序的可扩展性。比如，在“来自世界的问候”的这个例子中，如果我们增加一个来自新的国度的问候，所做的仅仅是声明 TMan 的新派生类，并创建它的对象。所以一旦用好多态，就能使自己的程序设计进入一个游刃有余的高妙境界。

5.7 用 VCL 的抽象类实现多态

Delphi 的 VCL 本身就有很多抽象类，这些抽象类为构建完美的 VCL 框架立下了汗马功劳。我们通过研究、学习 VCL 可以了解到面向对象的精髓，也能够编写出简洁优美的高质量代码。下面我以常用的流类（TStream）为例，来说明如何巧用 Delphi 抽象类实现多态编程。

我们观察流类，可以发现它有一些抽象方法，是一个不能实例化的抽象类：

```
TStream = class (TObject)
    .....
public
    function Read (var Buffer; Count: Longint): Longint; virtual; abstract;
    function Write (const Buffer; Count: Longint): Longint; virtual; abstract;
    .....
end;
```

virtual 和 abstract 限定符表明了 Read 和 Write 方法是虚方法。这说明 TStream 这个类并不能被真正使用（不能创建该类的实例），它只是一个类似于接口的类，它定义了作为 TStream 类应当具备以及需要处理的基本功能。而且它还规定了其他从 TStream 类派生出的类必须实现的功能（如 Read 以及 Write 等）。

流类可以将数据以流的形式保存到不同的存储介质中（磁盘文件、内存、数据库等），在日常编程中，我们需要使用到许多种具有流化能力的类，比如：读写文件的 TFileStream、读写内存的 TMemoryStream 以及读写数据库中二进制大对象字段的 TBlobStream。它们都是 TStream 的派生类。也就是说，TFileStream 以磁盘文件应用的方式实现了 TStream 类；而 TMemoryStream 则以内存应用的方式实现了 TStream 类。

现在假设我们要把数据流读入到一个 TMemo 控件中显示出来，TMemo 控件提供了一个很

好的 TMemol.Lines.LoadFromStream 方法。我们可能会这么写：

```
procedure TForm1.Button1Click (Sender: TObject);
var MyStream: TFileStream; //声明 TFileStream 类型的变量
begin
  MyStream: = TFileStream.Create ('计算机英语.txt', fmOpenRead);
  try
    Memol.Lines.LoadFromStream (MyStream);
  finally
    MyStream.Free;
  end;
end;
```

但是更好的写法是：

```
procedure TForm1.Button1Click (Sender: TObject);
var MyStream: TStream; //这里有变化,声明 TStream 类型的变量
begin
  // 此处 MyStream 的真正实例类型是 TFileStream,而不是 TStream
  MyStream: = TFileStream.Create ('计算机英语.txt', fmOpenRead);
  try
    Memol.Lines.LoadFromStream (MyStream);
  finally
    MyStream.Free;
  end;
end;
```

小小的改动体现了多态的思想。因为这里的对象变量定义的是 TStream 类型，虽然作为抽象类自身不能实例化，但它可以随时转型，适应不同类型的流，而无需改动其他代码。比如在下面的示例程序 5-7 中，我们可以用一条 Memol.Lines.LoadFromStream 语句把 TFileStream、TMemoryStream 和 TBlobStream 数据流读入到一个 TMemo 控件中，从而实现多态（见图 5-7）。

示例程序 5-7 巧用 TStream 抽象类

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  'Dialogs, StdCtrls, ExtCtrls, DB, Grids, DBGrids, DBTables;

type
  TForm1 = class (TForm)
    Memol: TMemo;
    RadioGroup1: TRadioGroup;
    Button1: TButton;
    Button2: TButton;
    Table1: TTable;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Table1Notes: TMemoField;
    Table1SpeciesNo: TFloatField;
    Table1Category: TStringField;
```

```

    Table1Common_Name: TStringField;
    Table1SpeciesName: TStringField;
    Table1Lengthcm: TFloatField;
    Table1Length_In: TFloatField;
    Table1Graphic: TGraphicField;
    procedure Button1Click (Sender: TObject);
    procedure CreateTStream (selectedIndex: integer);
    procedure Button2Click (Sender: TObject);
private
    FStream: TStream; //私有数据成员 FStream 定义为 TStream 类型
public
    end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.CreateTStream (selectedIndex: integer);
begin
    case selectedIndex of
        //将“计算机英语.txt”(文件中)读入到流中
        0: FStream := TFileStream.Create ('计算机英语.txt', fmOpenRead);
        //将 RadioGroup1 显示的文字项(内存中)读入到流中
        1: begin
            FStream := TMemoryStream.Create;
            RadioGroup1.Items.SaveToStream (FStream);
            FStream.Position := 0;
            end;
        //将数据库备注字段内容(Blob 字段中)读入到流中
        2: FStream := TBlobStream.Create (Table1Notes, bmRead);
    end;
end;

procedure TForm1.Button1Click (Sender: TObject);
begin
    CreateTStream (RadioGroup1.ItemIndex);
    try
        Memo1.Lines.LoadFromStream (FStream);
    finally
        FStream.Free;
    end;
end;

procedure TForm1.Button2Click (Sender: TObject);
begin
    Close;
end;

end.

```

大家可以在随书光盘中找到示例程序 5-7 的全部源代码。我们选择不同的流类型，可以看

到利用流技术从文件、内存和数据库字段等不同媒介读取并显示数据的情形，如图 5-7 所示。

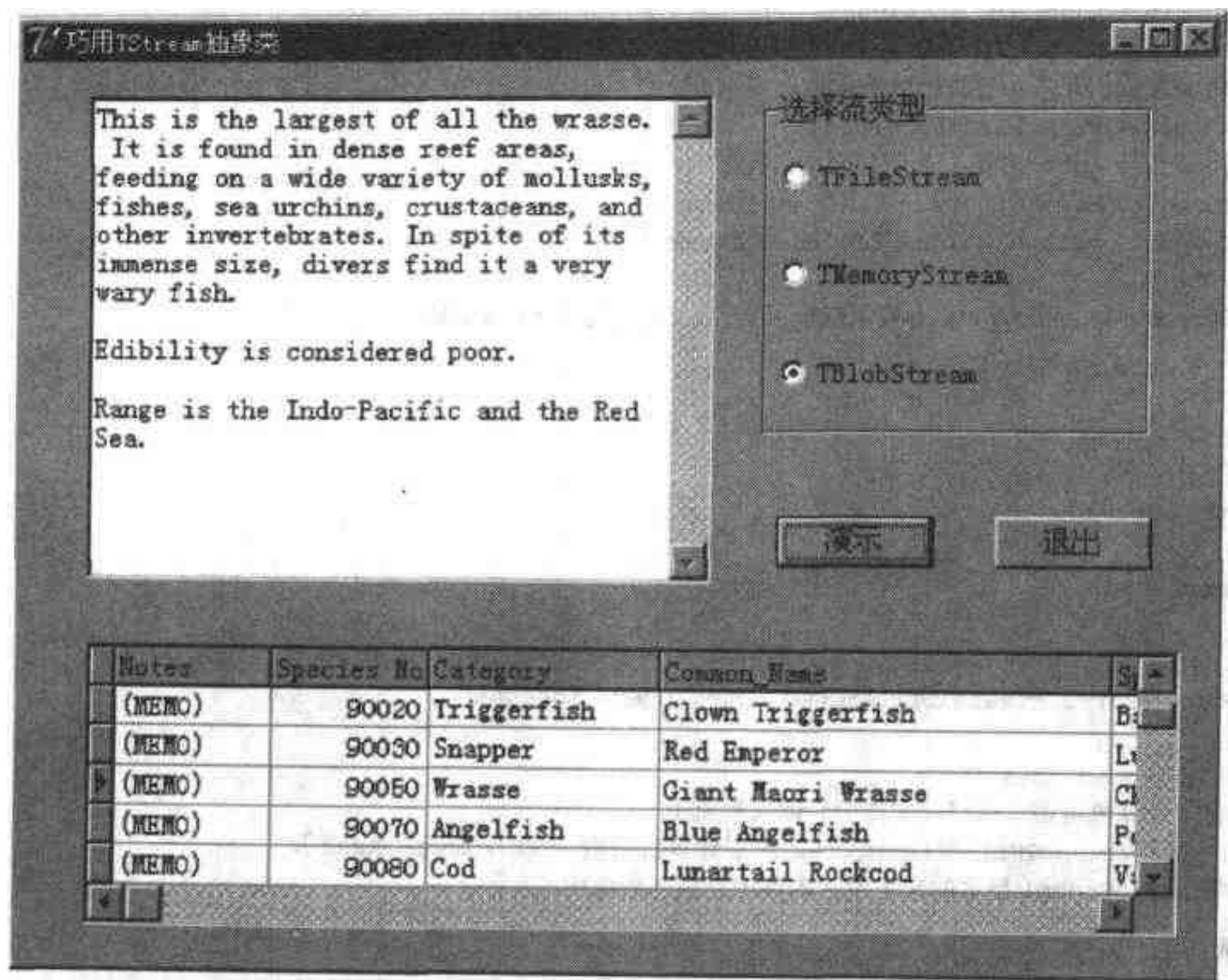


图 5-7 “巧用 TStream 抽象类” 程序利用流读取数据库备注字段时的效果

Delphi 中还有很多类似于 TStream 的抽象基类，如：TStrings。

```
TStrings = class (TPersistent)
.....
Protected
.....
function Get (Index: Integer): string; virtual; abstract;
function GetCount: Integer; virtual; abstract;
public
.....
procedure Clear; virtual; abstract;
procedure Delete (Index: Integer); virtual; abstract;
procedure Insert (Index: Integer; const S: string); virtual; abstract;
.....
end;
```

TStrings 为字符串的集合定义了一个接口。它在派生类中所实现的行为包括名字与值对的管理、将多个字符串作为一个连续的字符串处理、字符串搜索以及字符串表与其他流和外部文件之间的流化能力。在使用中，我们也应该将变量声明为 TStrings，然后实例化 TStrings 的派生类，并将其赋予 TStrings 变量。TStrings 的派生类包括：TStringList、TMemoStrings、TListBoxStrings 等。

TCustomIniFile 也是很有用的抽象基类，它的派生类有 TIniFile、TMemIniFile、TRegistryIniFile

等。令人惊奇的是 TRegistryIniFile 允许你用访问 Ini 文件的方式来访问注册表！这使得我们可以仿照示例程序 5-7，用一套代码实现写注册表和写 Ini 文件的功能。这其中的技术虽然简单，但是它的意义非同凡响！

在实际开发中，多态的使用主要是体现在设计类的构架中。这时我们要充分考虑多态，提供实现某种功能的中间类（抽象类）。类的规划很重要，在面向对象编程的时代，类的框架很大程度上决定了程序的框架，决定了软件开发的成败。结构清楚、层次分明的类构架，不仅易于功能划分与扩展，同时也更易于代码的维护。而其中，应用继承和多态的思想，引入抽象类，引入中间类，是较为可取的一种方法。

抽象类作为中间类对我们灵活使用 VCL 组件也是有帮助的，我们要以面向对象的思维去理解和运用它们，这个工作体现在定义一些过程、函数的参数上。

第6章 剖析接口

6.1 认识接口

从类继承是一个强大的机制，但更强大的继承来自于从一个接口的继承。接口是继承存在的真正原因；接口之所以重要，是因为一个接口允许用户完全分离方法名称与方法实现。到现在为止，还没有其他办法可以实现这一点（虚方法必须有一个主体）。接口所提供的分离非常强大，它允许用户真正分离“做什么”和“如何做”。接口只告诉用户方法的名称是什么，它并不关注是如何实现的。接口表示想要一个对象如何使用，而不是它在此刻是如何实现的。就好像我们每天使用很多设备，但并不知道它是如何工作的。例如，电视机的内部工作原理对于普通人并不重要，因为他们关心的只是如何使用它，而不是如何设计和制造它。所以，电视机的接口——遥控器上的按钮对用户才是最重要的。

6.1.1 什么是接口

这里我们要讨论的接口不仅仅是一种概念和方法，更是一种面向对象编程语言技术。前面我们用抽象方法和抽象类来实现接口技术，现在我们要研究的是真正意义上的接口，即 Delphi 语言中的一种叫做接口（interface）的类型，在 Delphi 中也称做对象接口（Object Interface）。

接口定义了能被一个类实现的方法。接口声明和类相似，但不能直接实例化，也不能自己实现它们的方法，而是由支持接口的类来提供实现。一个接口类型的变量能引用一个实现了该接口的对象，但是，只有接口中声明的方法才能通过该变量进行调用。

接口提供了一些多继承的好处，却没有多继承带来的语义困难。它们对使用分布式对象模型（如：COM+）也非常有效，支持接口的对象可以和其他语言编写的对象进行交互。

Delphi 对接口的定义提供了一种与封装、继承、多态性和方法覆盖同等重要的能力。然而对这种能力至今仍没有一个标准的术语称呼它，有的书上称之为等态性（isomorphism），这是一种把来自完全不相关类的若干对象统一处理的能力。

Delphi 接口确实简单，但又异常强大。Delphi 接口可用来定义对象所执行的一组方法。从类定义的起点出发，接口指定了类提供哪些方法和属性，但用户也可以定义一个使用接口的变量。正如用类名定义一个实例一样，当这样做时，变量可以拥有执行所指定接口的任何类的实例，比如，前面提到的“来自世界的问候”那个例子中，如果使用的是接口而不是抽象方法，SayHello 方法是 Greetable 接口的一部分，那么用户可以定义一个 Greetable 类型的变量，它可以含有类 TChinese、TAmerician 或 TFrench 的一个实例。从按此途径使用一个对象的起点出发，接口指定了使用该对象的代码可以应用哪些方法和数据。为了帮助用户更好地理解这些内容，让我们看一个对象是如何被使用的。

6.1.2 使用对象

假设用户在编写一个赛跑程序。赛跑程序含有几个对象，包括一个 Car（赛车）对象、一

个 Course (赛道) 对象及一个 Weather (天气) 对象。程序含有可以从这些对象中提取的数据, 调用它们的方法来管理比赛。

程序除了有代表赛道和天气的对象外, 还有一个称为 raceCar 的对象, 它是 Car 类的实例。程序赋予 raceCar 对象诸如 speedup、slowdown、pass 和 draft 之类的方法。

下面是用 Delphi 代码编写的赛车程序的粗线条轮廓。它显示在对象名和属性或方法间加一点号 (.) 的传统写法, 如 raceCar.speedup。

```
If on_straightaway then raceCar.speedup;  
If curve_ahead then raceCar.slowdown;  
If car_ahead_going_slower then racecar.pass;  
If car_ahead_at_same_speed then racecar.draft;
```

如果进一步抽象, 则我们可以用一种方法来概括这个程序, 实际上就是使用多态的解决方法。现在我们不用 Car 类的一个对象, 而是用 Vehicle (车辆) 类的派生对象。只要所有的 Vehicle 类的派生类都提供了正确的方法, 赛跑程序就可以用来比赛摩托车、自行车, 甚至还可以比赛卡车。

下面是重写的代码, 其中 raceVehicle 为 Vehicle 类的实例。现在它应用于比赛自行车以及小汽车:

```
if on_straightaway then raceVehicle.speedup;  
if curve_ahead then raceVehicle.slowdown;  
if vehicle_ahead_going_slower then raceVehicle.pass;  
if vehicle_ahead_at_same_speed then raceVehicle.draft;
```

但要注意前面我们强调的是“只要所有的 Vehicle 类的派生类都提供了正确的方法”; 可是并没有措施保证它们都能这样做! 由于卡车通常不用来比赛, 因此在 Truck 类中可能就没有定义 draft 方法。由于在 Vehicle 类中也没有定义 draft 方法, 结果 Truck 对象将根本没有 draft 方法, 这个程序将运行失败。

6.1.3 接口的引入

解决上面问题的一种办法是保证参赛的对象执行恰当的方法。在 Delphi 中, 可通过一个接口来达到。接口并不能实际完成一系列方法, 它只是指定了这些方法的存在, 如图 6-1 所示。

拥有 Raceable 接口的 Bicycle 和 Car 必须实现该接口指定的所有方法。它们已经从 Vehicle 类中继承了 speedup 和 slowdown 方法, 所以它们只需执行 pass 和 draft 方法来成为 Raceable 对象。这样对于早先的程序, 仅通过将 raceVehicle 对象修改为 Raceable 对象就可以了。

对 RaceDriver 程序的惟一改变就是 raceVehicle 对象定义为 Raceable 类型而不是 Vehicle 类型, 即变量声明从 raceVehicle: TVehicle 变成了 raceVehicle: IRaceable, 其他东西都没有变, 甚至连代码都是一样的:

```
if on_straightaway then raceVehicle.speedup ();  
if curve_ahead then raceVehicle.slowdown ();  
if vehicle_ahead_going_slower then raceVehicle.pass ();  
if vehicle_ahead_at_same_speed then raceVehicle.draft ();
```

但是这一个改变却带来了很大变化, 因为现在的赛跑程序适用于许多种不同的 Raceable 对

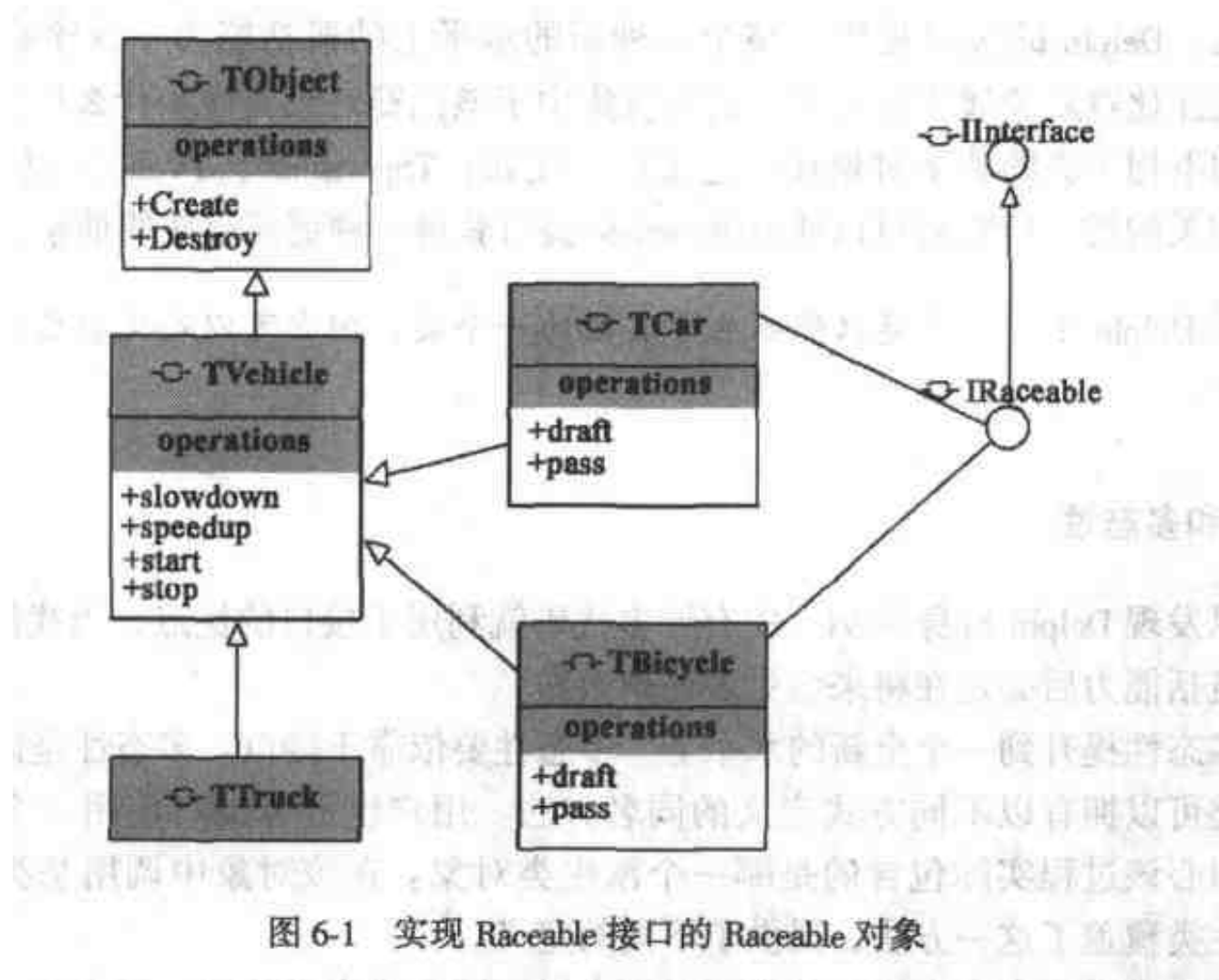


图 6-1 实现 Raceable 接口的 Raceable 对象

象了。

接口的力量源自于它是作为数据类型使用的这一事实。任何实现 Raceable 接口的类都可以用于赛跑程序（实际上已经不是原来的赛车程序了）。这意味着同样的程序可以用于 RaceHorse、Skater、Sailboat 或 Runner 这些类型的比赛对象，只要这些类能够执行 speedup、slowdown、pass 和 draft 方法即可，如图 6-2 所示。

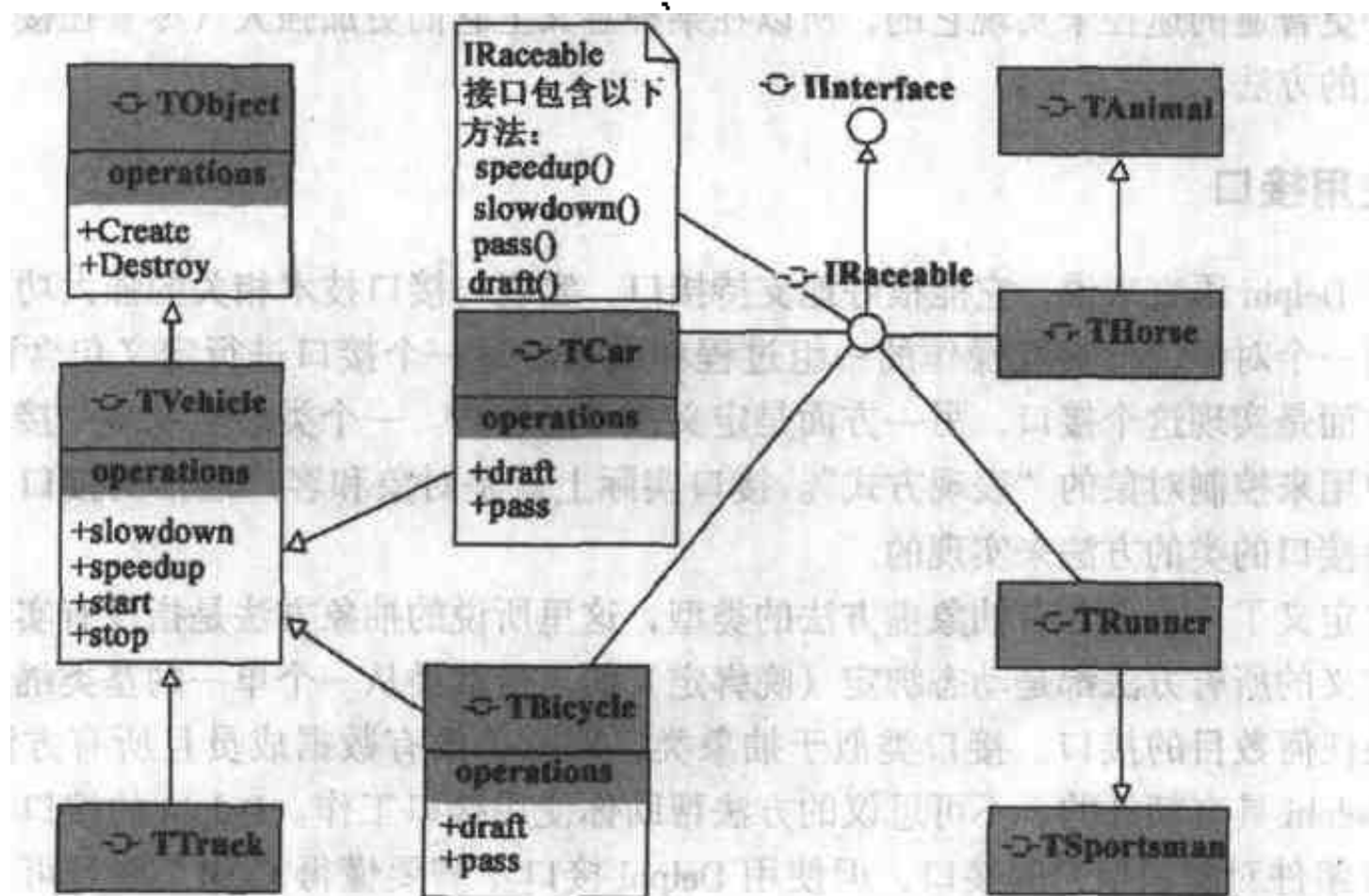


图 6-2 任何实现 Raceable 接口的类都可以用于赛跑程序

显而易见, Delphi 的接口提供了整个一种新的水平上的概括能力。程序有了多得多的自由。它不再关注比赛对象属于什么类,而仅仅集中于该比赛对象要做些什么!使用这种方法允许来自广泛的不相关类的若干对象在一起工作。比如: `TSportsman`、`TVehicle` 以及 `TAnimal` 就是这样一些不相关的类,但它们可以通过 `Raceable` 接口获得一种更高层次的抽象。

提示 在 Delphi 中,一个类只能派生于其他的一个类,但它可以定义自己需要的任意多的接口。

6.1.4 接口和多态性

读者可以发现 Delphi 自身的 VCL 中有许多代码就利用了接口的优点,当我们学会利用接口提供的强大概括能力后必定在将来会更多地这么做。

接口把多态性提升到一个全新的水平上。多态性要依靠于接口。多态性是说派生于同一个基类的两个类可以拥有以不同方式定义的同名方法。用户也可以编写使用一个基类对象的过程,而不必担心该过程实际包含的是哪一个派生类对象。在该对象中调用基类的方法仍然适用,不过派生类覆盖了这一方法,提供了不同的定义。

然而,接口是说两个完全无关的类可以拥有以不同方式定义的同一组方法。用户可以编写使用一个接口对象的过程,而不必担心此过程包含的是哪一个对象。调用该类的对象上的接口方法仍然适用,尽管每个类只提供了该方法的一个不同的实现。

在以上两种情况下,程序都获得了在一种抽象水平上处理对象集的能力,而让每个对象自己去关注具体的细节。基类和接口都确保了能够定义程序员所能依赖的一组特定方法。接口是通过一种更普遍的途径来实现它的,所以在某种意义上它们更加强大(尽管在接口用户不能继承已存在的方法)。

6.2 使用接口

对于 Delphi 语言来说,它能很好地支持接口,实现与接口技术相关的强大功能。接口定义了能够与一个对象进行交互操作的一组过程和函数。对一个接口进行定义包含两个方面的内容,一方面是实现这个接口,另一方面是定义接口的客户。一个类能实现多个接口,即提供多个让客户用来控制对象的“表现方式”。接口实际上就是对象和客户通信的接口,所以接口是通过支持接口的类的方法来实现的。

接口定义了一个包含有抽象虚方法的类型,这里所说的抽象方法是指没有实现的方法,因为接口定义的所有方法都是动态绑定(晚绑定)的。虽然类从一个单一的基类继承而来,但它可以实现任何数目的接口。接口类似于抽象类,即一个没有数据成员且所有方法都是抽象的类,但 Delphi 具有额外的、不可思议的方法帮助你使用接口工作。Delphi 的接口有时候看起来像 COM(组件对象模型)的接口,但使用 Delphi 接口不需要懂得 COM,而且可以用接口来达到许多其他的目的。

6.2.1 定义接口

要声明一个接口，使用的是限定符 `interface` 而不是 `class`。在接口中，就像在类中一样声明方法，只不过不需要指定访问限定符，如：`public`、`private`、`protected` 等，因为作为接口的方法，总是 `public` 的。通常接口命名时使用大写 `I` 作为前缀，为便于理解，建议命名为 `XXXable`。

可以通过继承一个已有的接口来声明一个新接口，就像继承一个已有的基类来声明一个新类一样。接口声明包含方法和属性的声明，但不包括数据成员。和所有的类都从 `TObject` 继承而来一样，所有的接口都继承自 `IInterface` 接口。`IInterface` 接口声明了三个方法：`_AddRef`、`_Release` 和 `QueryInterface`。如果熟悉 COM，你就会知道这些方法。前两个方法为实现该接口对象的生命周期管理其引用计数，从而为接口引用提供了生命期内存管理；第三个方法可以查询一个对象可能实现的其他接口，通过调用 `QueryInterface` 接口能够获得新接口的引用，从而支持在一个对象所实现的不同接口之间自由跳转。实现这三个方法最简单的方式是从 `TInterfacedObject`（在 `System` 单元声明）派生一个类。

`IInterface` 在 Delphi 7 的 `system` 单元中定义如下：

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface (const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
  IUnknown = IInterface;
```

需要说明的是，几乎所有 Delphi 书籍都说一切接口都继承自 `IUnknown` 接口。但是在 Delphi 7 中，为了跨平台的需要，改成了 `IInterface` 接口，但仍然兼容 `IUnknown` 接口，`IUnknown` 接口最好只用在依赖 Windows 平台的一些特殊程序中。从上面的定义代码中，我们显然可以看到这一变化。

提示 Borland 已经将 Delphi 的 `system` 单元改称为 `Delphi / Kylix Cross-Platform Runtime Library System Unit`。

接口的定义就像是类的定义，最根本的不同是在接口中有一个全局唯一标识符（GUID），它对于每一个接口来说是不同的。Delphi 对 `IInterface`（就是以前的 `IUnknown`）的定义来自于微软的组件对象模型（COM）规范，因此 Delphi 对 COM 的支持非常好。

由于每个接口在使用中潜在地存在命名多义性的问题；因此，它们需要指派全局唯一标识符。GUID 对接口的协同工作是很关键的，GUID 使接口可以在任何系统中使用而不会有名字冲突的可能。

提示 全局唯一标识符（GUID）使用由开放软件基金会（OSF, Open Software Foundation）分布式计算环境（DCE, Distributed Computing Environment）创建的算法生成。每一个值是 16 个字节（128 位）的全局唯一标识符（UUID）。UUID 是由开放软件

基金会 (OSF) 使用的最初的术语, 但自从出现了 GUID 后, 两个术语可互换使用。用做生成 UUID 的算法使用 48 位网络适配器的 ID 和当前的日期及在最近的 100 纳秒间隔内从 1582 年 10 月 15 日的子夜 12 时开始的时间。这种方法使两个 GUID 是同一个值的可能性非常地小。

声明一个接口如同声明一个 Delphi 的类, 所以自己定义一个接口并不难。接口只能像类一样在程序或单元的最外层声明, 一个接口的声明格式如下:

```
type interfaceName = interface (ancestorInterface)
    ['{GUID}']
    memberList
end;
```

这里, (ancestorInterface) 和 ['{GUID}'] 是可选的。在大多数方面, 接口声明和类声明相似, 但有以下限制:

- memberList 只能包含方法和属性, 而不能包含数据成员 (field)。
- 因为接口没有数据成员, 所以在接口中属性的读 (read) 和写 (write) 限定符必须是方法。
- 接口的所有成员都是公有的 (public), 不允许使用可见性限定符。
- 接口没有构造函数和析构函数, 它们不能直接被实例化, 除非使用实现了它们方法的类。
- 方法不能被声明为 virtual、dynamic、abstract 或 override。因为接口自己不实现它们的方法, 这些声明没有意义。

下面的示例代码定义了一个新的接口 INew, 它包含一个被称为 F1 的方法:

```
type
    INew: interface
        ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
        function F1: Integer;
    end;
```

提示 在 Delphi 的 IDE 中, 按 Ctrl + Shift + G 键可以为一个接口生成一个新的 GUID。

下面的代码声明了一个称为 IFromNew 的接口, 它是从 INew 接口继承来的:

```
type
    IFromNew = interface (INew)
        ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
        function F2: Integer;
    end;
```

6.2.2 实现接口

要实现一个接口, 需要声明一个从该接口继承的类, 并实现该接口方法。同样, 要声明一个实现多个接口的类时, 必须提供在所有接口中声明的所有方法的实现。这种多接口的实现类似多重继承的功能。声明的格式如下:

```

type className = class (ancestorClass, interface1, ..., interfaceN)
  memberList
end;

```

类可以实现一个接口的方法，或将一个接口赋值给一个属性。但是并不是任何类都可以实现接口（除非该类中已经有支持 IInterface 接口或它的派生接口的实现，并保证实现了 _AddRef、_Release 和 QueryInterface 方法）。

以下程序是无法实现 IGreetable 接口的，因为 TMan 继承自一个不支持接口的类 TObject。Delphi 编译时将提示出错信息：“[Error] Unit1.pas (16): Undeclared identifier: 'QueryInterface'”，因为 TObject 中根本就没有声明 QueryInterface 方法，更谈不上为接口提供 QueryInterface 的实现了。

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  IGreetable = interface
    ['{FE5A34E5-21AB-4120-971B-FDC3241AD55D}']
    function SayHello: string;
  end;

  TMan = class (TObject, IGreetable)
    function SayHello: string;
  end;

  TForm1 = class (TForm)
    Button1: TButton;
    procedure Button1Click (Sender: TObject);
  private
    procedure Greeting (Intf: IGreetable);
    {Private declarations}
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

function TMan.SayHello: string;
begin
  Result: = 'Hello';
end;

procedure TForm1.Greeting (Intf: IGreetable);

```

```

begin
    ShowMessage (Intf.SayHello);
end;

procedure TForm1.Button1Click (Sender: TObject);
var
    Intf: IGreetable;
begin
    Intf := TMan.Create;
    Greeting (Intf);
end;

end.

```

要实现 `_AddRef`, `_Release` 和 `QueryInterface` 方法的最简单办法是从 `TInterfacedObject` 或它的一个派生类继承它们。以下程序就可以实现 `IGreetable` 接口, 因为 `TMan` 继承自一个 `TInterfacedObject` 类。

```

TMan = class (TInterfacedObject, IGreetable)
    function SayHello: string;
end;

```

`TInterfacedObject` 类何以有如此神奇的功能, 让我们来看看其“庐山真面目”。在 Delphi 7 的 `system` 单元找到它的声明, 如下所示:

```

TInterfacedObject = class (TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface (const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
end;

```

可以发现 `TInterfacedObject` 也是继承自 `TObject` 和 `IInterface`, 不过它提供了 `IInterface` 实现 `_AddRef`、`_Release` 和 `QueryInterface` 的方法。如果你希望自己定义这些方法, 也可以自由地从任何其他类继承过来。下面我们不妨一试。

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
    IGreetable = interface
        ['FE5A34E5-21AB-4120-971B-FDC3241AD55D']
        function SayHello: string;
    end;

```

```
end;

TMan = class (TObject, IGreetable)
  function SayHello: string;
  function QueryInterface (const IID: TGUID; out Obj): HRESULT; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;

TForm1 = class (TForm)
  Button1: TButton;
  procedure Button1Click (Sender: TObject);
private
  procedure Greeting (Intf: IGreetable);
  {Private declarations }
public
  {Public declarations }
end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

// 这里实现自己定义的 _AddRef、_Release 和 QueryInterface 方法
function TMan.QueryInterface (const IID: TGUID; out Obj): HRESULT;
begin
  if GetInterface (IID, Obj) then
    Result: = 0
  else
    Result: = Windows.E_NoInterface;
end;

function TMan._AddRef: Integer;
begin
  Result: = -1
end;

function TMan._Release: Integer;
begin
  Result: = -1
end;

function TMan.SayHello: string;
begin
  Result: = 'Hello';
end;

procedure TForm1.Greeting (Intf: IGreetable);
begin
  ShowMessage (Intf.SayHello);
end;
```

```

procedure TForm1.Button1Click (Sender: TObject);
var
    Intf: IGreetable;
begin
    Intf := TMan.Create;
    Greeting (Intf);
end;

end.

```

如果是下面这样一个程序，TMan 继承自 TEdit，能不能运行呢？

```

unit Unit1;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
    IGreetable = interface
        ['{FE5A34E5-21AB-4120-971B-FDC3241AD55D}']
        function SayHello: string;
    end;

    TMan = class (TEdit, IGreetable) //注意这里的变化
        function SayHello: string;
    end;

    TForm1 = class (TForm)
        Button1: TButton;
        procedure Button1Click (Sender: TObject);
    private
        procedure Greeting (Intf: IGreetable);
        {Private declarations}
    public
        {Public declarations}
    end;

var
    Form1: TForm1;

implementation
{$R *.DFM}

function TMan.SayHello: string;
begin
    Result := 'Hello';
end;

procedure TForm1.Greeting (Intf: IGreetable);
begin
    ShowMessage (Intf.SayHello);
end;

```

```

procedure TForm1.Button1Click (Sender: TObject);
var
  Intf: IGreetable;
  obj: TMan;
begin
  Intf := TMan.Create (self);
  Greeting (Intf);
end;

end.

```

当然可以，因为从 Delphi 6 开始 TComponent 继承了 IInterface。

```
TComponent = class (TPersistent, IInterface, IInterfaceComponentReference)
```

并且提供了 IInterface 实现 _AddRef、_Release 和 QueryInterface 的方法：

```
|TComponent.IInterface|
```

```

function TComponent.QueryInterface (const IID: TGUID; out Obj): HRESULT;
begin
  if FVCLComObject = nil then
  begin
    if GetInterface (IID, Obj) then Result := S_OK
    else Result := E_NOINTERFACE
  end
  else
    Result := IVCLComObject (FVCLComObject).QueryInterface (IID, Obj);
end;

```

```

function TComponent._AddRef: Integer;
begin
  if FVCLComObject = nil then
    Result := -1 // -1 indicates no reference counting is taking place
  else
    Result := IVCLComObject (FVCLComObject)._AddRef;
end;

```

```

function TComponent._Release: Integer;
begin
  if FVCLComObject = nil then
    Result := -1 // -1 indicates no reference counting is taking place
  else
    Result := IVCLComObject (FVCLComObject)._Release;
end;

```

类可以通过相同的名称、参数和调用规范声明来实现一个接口所包含的每一个方法。Delphi 自动将类的方法与接口的方法匹配起来。如果想用一个不同的方法名称，就可以将接口方法重定向到一个不同名称的方法。被重定向的方法必须具有与接口方法相同的参数和调用规范。当一个类以同样的方法名称实现多个接口时，这一特性就显得特别重要了。

下面的代码演示了在一个类 TFootball 中怎样实现 IFoot 和 IBall 接口：


```

type
  TFootball = class (TInterfacedObject, Ifoot, IBall)
    function F1: Integer;
    function F2: Integer;
  end;

implementation

function Tfootball.F1: Integer;
begin
  Result: = 0;
end;

function Tfootball.F2: Integer;
begin
  Result: = 9;
end;

```

注意 一个类可以实现多个接口，只要在声明这个类时依次列出要实现的接口即可。编译器通过名称来把接口中的方法与实现接口的类中的方法对应起来，如果一个类只是声明要实现某个接口，但并没有具体实现这个接口的方法，编译将出错。

如果一个类要实现多个接口，而这些接口中包含同名的方法，则必须把同名的方法另取一个别名，请看下面的程序示例：

```

type
  IFoot: interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;
,
  IBall = interface
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    function F1: Integer;
  end;

  TFootball = class (TInterfacedObject, IFoot, IBall)
    //为同名方法取别名
    function IFoot.F1: FootF1;
    function IBall.F1: BallF1;
    //接口方法
    function FootF1: Integer;
    function BallF1: Integer;
  end;

implementation

function TFootball.FootF1: Integer;
begin
  Result: = 0;
end;

function TFootball.BallF1: Integer;

```

```
begin
    Result := 0;
end;
```

Delphi 还使用 `implements` 指示符用于委托另一个类或接口来实现接口的某个方法，这个技术有时又被称为委托实现，关于 `implements` 指示符的用法，请看下面的代码：

```
type
    TSomeClass = class (TInterfacedObject, IFoot)
        .....
        function GetFoot: TFoot;
        property Foot: TFoot read GetFoot implements IFoot;
        .....
    end;
```

在上面例子中的 `implements` 指示符是要求编译器在 `Foot` 属性中寻找实现 `IFoot` 接口的方法。属性的类型必须是一个类，它包含 `IFoot` 方法或类型是 `IFoot` 的接口或 `IFoot` 派生接口。`implements` 指示符后面可以列出几个接口，彼此用逗号隔开。

类可以将一个接口的实现指派给一个使用 `implement` 指示字的属性。该属性的值必须是类希望实现的接口。当对象被指派给该接口类型时，Delphi 自动取得属性的值并返回该接口。

`implements` 指示符在开发中提供了两个好处：首先，它允许以无冲突的方式进行接口聚合。聚合（aggregation）是 COM 中的概念。它的作用是把多个类合在一起共同完成一个任务。其次，它能够延后占用实现接口所需的资源，直到确实需要资源。例如，假设实现一个接口需要分配 1MB 位图的资源，但这个接口很少用到。因此，可能平时你不想实现这个接口，因为它太耗费资源了，用 `implements` 指示符后，可以只在属性被访问时才创建一个类来实现接口。

当在应用程序中使用接口类型的变量时，要用到一些重要的语法规则。最需要记住的是，一个接口是生存期自我管理类型的。这意味着，它通常被初始化为 `nil`。接口是引用计数的，当获得一个接口时自动增加一个引用计数；当它离开作用域或赋值为 `nil` 时就被自动销毁。

因为 Delphi 编译器产生对 `_AddRef` 和 `_Release` 的调用来管理接口对象的生命周期，所以，为了使用 Delphi 的自动引用计数，可以一个接口类型声明一个变量。当分配一个接口引用给一个接口变量时，Delphi 就自动调用 `_AddRef`。当该变量超出范围时，Delphi 就自动调用 `_Release`。`_AddRef` 和 `_Release` 的行为完全由用户控制。如果是从 `TInterfacedObject` 继承而来的，这些方法将实现引用计数。`_AddRef` 方法增加引用数，`_Release` 则减少它。当引用数变为零的时候，`_Release` 就销毁对象。如果是从一个不同的类继承而来的，则可以定义这些方法来做任何希望做的事情。但是，应当正确地实现 `QueryInterface`，因为 Delphi 依靠它来实现 `as` 运算符。

下面的代码演示了一个接口变量的生存期自我管理机制。

```
var
    I: ISomeInterface;
begin
```

```

//I 被初始化为 nil
I := FunctionReturningAnInterface; //I 的引用计数加 1
I.SomeFunc;
//I 的引用计数减 1, 如果为 0, 则自动销毁。
end;

```

6.3 接口与抽象类

在特定情况下, 抽象类可以看做是一个接口。抽象类不能直接用来生成对象, 所以它的真实意图是用做它的派生类的一个接口。读者在示例程序 5-3 用抽象方法修改后的 uSayHello 程序中已经看到过抽象类的例子, 其中抽象方法 SayHello 被用做为接口, 因为在抽象类 TMan 中根本就没有实现 SayHello 方法的任何代码, SayHello 方法作为一个接口是留给派生类实现的。

接口也可以看做是包含抽象方法的最终抽象类, 甚至在程序写法上接口与只包含抽象方法的抽象类也有相似之处。

一个类可以实现一个接口。如此一来, 类就要实现接口的方法, 与派生类保证要实现它的基类的抽象方法一样。所以可以把接口看做是动态绑定函数调用抽象基类的一种替代方式。

为了让读者深刻理解接口与抽象类的关系, 下面用接口编程的方式重新改写“来自世界的问候”那个例子。

在示例程序 6-1 中, 可以看到, 我首先在 uSayHello 中新增了一个接口 IGreetable, 并用该接口的 SayHello 方法来替代抽象类 TMan 中的原有抽象方法 SayHello。这不会影响我们通过动态绑定调用 SayHello 方法的能力。TChinese、TAmerican、TFrench 和 TKorean 类继承了 IGreetable 接口并实现了 SayHello 方法。实际上, 实现 SayHello 方法的代码和原先一样, 我们无需做任何改动。

示例程序 6-1 用接口编程方式修改过的 uSayHello

```

unit uSayHello;

interface

type

    IGreetable = interface
        ['{D91DDE09-0FC4-4FE9-AE0D-9877E2F73BF6}']
        function SayHello: string;
    end;

    TMan = class (TInterfacedObject)
        Language: string;
        Married: Boolean;
        Name: string;
        SkinColor: string;
    public
        constructor create; virtual;

```

```
end;

TChinese = class (TMan, IGreetable)
public
    constructor create; override;
private
    function SayHello: string;
end;

TAmerican = class (TMan, IGreetable)
public
    constructor create; override;
private
    function SayHello: string;
end;

TFrench = class (TMan, IGreetable)
public
    constructor create; override;
private
    function SayHello: string;
end;

TKorean = class (TMan, IGreetable)
public
    constructor create; override;
private
    function SayHello: string;
end;

implementation

constructor TMan.create;
begin
    Name: = '张三';
    Language: = '中文';
    SkinColor: = '黄色';
end;

constructor TChinese.create;
begin
    inherited;
end;

constructor TAmerican.create;
begin
    Name: = 'Lee';
    Language: = '英文';
    SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
    Name: = '苏菲';
```

```
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;

function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

end.
```

示例程序 6-1 和示例程序 5-3 相比较而言变动不是太大，但仍然有以下几方面需要注意：

- TMan 是从 TInterfacedObject 继承而来，而不是 TObject。这样就保证其派生类可以实现接口的方法。
- 由于实现的是接口的方法，而不是覆盖基类的抽象方法，因此派生类在声明 SayHello 时不再使用 override 限定符。
- 尽管接口和抽象类在句法和语义上紧密相关，但仍有一个重要区别：接口只能包含抽象方法，而抽象类除了包含抽象方法外，还能包含数据成员以及非抽象方法。所以，多重接口不会像多重继承那样存在着隐含冲突。结果是，在 Delphi 中一个类只能继承一个基类，但可以有无限多个接口。另外，接口和抽象类有时也不能互相取代。例如，你不能用 IGreetable 完全取代 TMan，因为 IGreetable 不能包含数据成员和有自己实现内容的非抽象方法 TMan.create。

前面，我在逻辑单元 uSayHello 中用接口改写了原来的程序，那么界面单元 ufmSayHello 中需要如何改写原来的代码呢？请看示例程序 6-2。

示例程序 6-2 为适应接口编程而修改过的 ufrmSayHello

```
unit ufrmSayHello;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, uSayHello;

type
  TfrmSayHello = class (TForm)
    GroupBox1: TGroupBox;
    edtName: TLabelledEdit;
    edtSkinColor: TLabelledEdit;
    edtLanguage: TLabelledEdit;
    btnUSA: TButton;
    btnKorean: TButton;
    btnCN: TButton;
    btnFrench: TButton;
    Image1: TImage;
    procedure btnUSAClick (Sender: TObject);
    procedure btnCNClick (Sender: TObject);
    procedure btnFrenchClick (Sender: TObject);
    procedure btnKoreanClick (Sender: TObject);
  private
    procedure sayhello (AMan: TMan; G: IGreetable);
  public
    {Public declarations}
  end;

var
  frmSayHello: TfrmSayHello;

implementation

{$R *.dfm}

procedure TfrmSayHello.sayhello (AMan: TMan; G: IGreetable);
begin
  //类实现的多态
  edtName.Text: = AMan.Name;
  edtLanguage.Text: = AMan.Language;
  edtSkinColor.Text: = AMan.SkinColor;
  //接口实现的多态
  image1.Picture.LoadFromFile (G.sayHello);
end;

procedure TfrmSayHello.btnUSAClick (Sender: TObject);
var
  G: IGreetable; AMan: TMan;
begin
  AMan: = TAmerican.create;
  G: = TAmerican.create;
```

```

    sayhello (AMan, G);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject);
begin
    sayhello (TChinese.create, TChinese.create);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject);
begin
    sayhello (TFrench.create, TFrench.create);
end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject);
begin
    sayhello (TKorean.create, TKorean.create);
end;

end.

```

在 `ufmSayHello` 中，程序主要的变动有两方面。首先派生类从基类 `TMan` 和接口 `IGreetable` 分别继承了属性和方法，也就是说这里出现了由类实现的多态和由接口实现的多态两种情况。于是我们需要修改 `TfrmSayHello.sayhello` 方法，分别传入 `AMan` 和 `G` 参数，然后分别去实现 `AMan` 对象的属性和 `G` 接口的方法。

另一方面，在按钮事件调用 `TfrmSayHello.sayhello` 方法时，我们会看到类似下面的写法：

```

procedure TfrmSayHello.btnCNClick (Sender: TObject);
begin
    sayhello (TChinese.create, TChinese.create);
end;

```

请注意在两个参数中，此 `TChinese.create` 非彼 `TChinese.create`，前者创建的是对象，后者创建的是接口。如果想不通，可以回顾一下 `TChinese` 的声明：

```

TChinese = class (TMan, IGreetable)
public
    constructor create; override; //来自 TMan
private
    function SayHello: string; //来自 IGreetable
end;

```

这是不是有点像多重继承？实际上为了好理解，也可以这样写：

```

procedure TfrmSayHello.btnCNClick (Sender: TObject);
var
    G: IGreetable; AMan: TMan;
begin
    AMan := TChinese.create;
    G := TChinese.create;
    sayhello (AMan, G);
end;

```

以上两种写法是等价的，不过它们都不是最好的写法。真正的高手在写程序时不仅要考虑

功能的正确实现，还要考虑对原代码的改动最小，使可重用性最大。

我们前面讲过类的类型转换问题。其实接口也可以实现类似的转换。实际上像 TChinese 这样的对象一旦创建，既包含了类的方法（如：create），也包含了接口的方法（如：SayHello）。所以，我们在 TfrmSayHello.sayhello 中只要传递一个类类型的参数 AMan: TMan 就可以了，在这个参数基础上，通过类型转换，就可以获得接口类型和接口方法。请看示例程序 6-3。

示例程序 6-3 通过类型转换，就可以获得接口类型和接口方法

```
procedure TfrmSayHello.sayhello (AMan: TMan);
var G: IGreetable;
begin
    //类实现的多态
    edtName.Text: = AMan.Name;
    edtLanguage.Text: = AMan.Language;
    edtSkinColor.Text: = AMan.SkinColor;
    //通过类型转换,就可以获得接口类型和接口方法
    G: = AMan as IGreetable;
    //接口实现的多态
    image1.Picture.LoadFromFile (G.sayHello);
end;

//其他代码无需改动
procedure TfrmSayHello.btnUSAClick (Sender: TObject);
begin
    sayhello (TAmerican.create);
end;
```

这是一种更简洁优美的写法，它使修改过的程序保持了与示例程序 5-3 一致的风格，即体现多态的那种风格。惟一的改动是利用 as 限定符在 TfrmSayHello.sayhello 中将变量由类类型转变成接口类型，以实现接口方法。如果你不深入了解接口，不知道一个接口变量与实现这个接口的类是赋值相容的这样一个接口变量规则，那么，别说写了，你可能都无法看懂这样的程序。

6.4 接口关系

Delphi 在其定义接口的能力方面是独一无二的，它收集所有的方法和属性（实际上是属性的 read 和 write 方法），这些特征能确保可以在任何实现接口的类中都是可用的。接口提供了很多设计上的灵活性，并且给类/对象增加了一些复杂性。事实上，理解了接口就容易理解基类/派生类的关系，特别是当基类是一个抽象类时更是如此。

6.4.1 类、对象和接口的关系

接口可以看做是某一个类的一个添加模板。当某个类实现某个接口时，它确保所有定义在接口中的方法都定义在类中，该接口定义了对对象的操作的一个子集。

因为接口指定了一组可能的交互动作，所以接口可以用做变量的数据类型（type）而不是一个类名称。令人高兴的是，可以将实现接口的任何类的一个实例赋给那个接口变量。当然，

它的局限性还在于只有在这个接口中定义的方法才可以在该对象上被予以调用。任何对象可能实现的其他方法是不可见的。例如，在示例程序 6-2 中，IGreetable 接口变量 G 只能实现 TMan 派生类中的 sayHello 方法，而不能实现其他方法，其他方法（如 create 方法）对 G 是不可见的。

```
procedure TfrmSayHello.sayhello (AMan: TMan; G: IGreetable);
begin
    //类的实现
    edtName.Text: = AMan.Name;
    edtLanguage.Text: = AMan.Language;
    edtSkinColor.Text: = AMan.SkinColor;
    //接口的实现
    image1.Picture.LoadFromFile (G.sayHello);
end;
```

作为类的一个实例，对象可以响应在任何（它所实现的）接口中定义的方法。但是作为接口的一个实例，它只能响应在那个接口中定义的方法。也就是说，接口标识类中方法的某个子集。

同样的道理，接口和类之间的区别还在于对象属性只能用类来访问。因为接口定义只包含方法和内容，它不可能用一个接口直接访问对象属性。

前面我们还比较了接口和抽象类，虽然两者有相似的用途，但接口并不是“更抽象”的抽象类。由于接口不带有任何实现（对数据的存储和操作），也就是说接口并不与任何存储空间有关联，所以继承（或合并）多个接口并不增加编译器的负担，这使得接口在实现多重继承上极具有优势。Delphi 不支持派生类同时继承多个基类，因为每个基类都包含具体的实现，而且也无法确保它们都属于相同的类型。Delphi 虽然采用了关系清晰、安全易用的单亲继承体系，但仍然可以用多接口的继承实现类似于多重继承的功能。

6.4.2 接口引用关系

如果声明一个接口类型的变量，则它可以引用任何实现这个接口的类实例。这样的变量使我们可以调用接口的方法，而不必在编译时知道接口是在哪里实现的。但要注意以下限制：

- 使用接口类型的表达式只能访问接口定义的方法和属性，而不能访问实现类的其他成员。
- 一个接口类型的表达式不能引用实现了它的派生类接口的类实例，除非这个类（或它的继承类）还明确实现了此祖先接口。

例如：

```
type
    IGreetable = interface
    end;

    IMan = interface (IGreetable)
        function SayHello: string;
    end;
```

```

TChinese = class (TInterfacedObject, IMan)
    procedure setChinese (name: string);
    function SayHello: string;
end;
...
var
    AMan: IMan;
    Greeting: IGreetable;
begin
    AMan := TChinese.create; //工作正常
    Greeting := TChinese.create; //错误
    AMan.setChinese ('张三'); //错误
    AMan.SayHello; //工作正常
end;

```

在这个例子中：

- Greeting 被声明为 IGreetable 类型的变量，因为 TChinese 声明实现的接口中没有列出 IGreetable，所以 TChinese 类型的实例不能赋给 Greeting。但如果改变 TChinese 的声明为：

```

TChinese = class (TInterfacedObject, IGreetable, IMan)
...

```

那么，第一个错误语句将变得可用：

```

(Greeting := TChinese.create; )

```

- AMan 被声明为 IMan 类型的变量，虽然它可以引用 TChinese 类型的实例，但我们不能用它访问 TChinese 的 setChinese 方法，因为该方法不是 IMan 接口的方法。但如果改变 AMan 的声明为：

```

AMan: TChinese;

```

则第二个错误语句将变得可用：

```

(AMan.setChinese ('张三'); )

```

如果要判断一个接口类型的表达式是否引用了一个对象，则可通过标准函数 Assigned 来完成。

6.4.3 互相依赖的接口

不同接口之间可以在自己的方法中互相调用对方，形成依赖关系。但是，相互继承（派生）的接口是不允许的，比如，IControl 派生 IWindow，又从 IWindow 派生 IControl 是非法的。如果要定义互相依赖的接口，需要提前声明没有定义的接口，即使用一个 interface 关键字定义这个接口。一个 Forward 声明，无需指出接口的祖先、GUID 以及成员列表，它仅仅是一个事先的类型声明而已。例如：

```

type
    IControl = interface; // IControl 的 Forward 声明
    IWindow = interface
        ['{00000115-0000-0000-C000-000000000044}']
        function GetControl (Index: Integer): IControl;
        //如果没有 IControl 的 Forward 声明, GetControl 函数返回 IControl 类型就是非法的;

```

```

.....
end;
IControl = interface//IControl 的实际声明
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
.....
end;

```

6.4.4 接口与类型转换

一个类和它实现的任何接口是赋值兼容的，一个接口和它的祖先接口是赋值兼容的。另外，可以赋值给接口变量的还有 nil，把 nil 赋值给接口变量能够让 Delphi 在该变量的旧值上调用 _Release（如果该变量不是从 nil 开始的话）。对于一个接口类型的表达式，可以被赋予一个变体类型（Variant），如果接口类型是 IDispatch，则 Variant 变量的类型码是 varDispatch，否则为 varUnknown。

对于变量和值类型转换，接口类型和类类型遵循同样的原则。若一个类实现了某个接口，则类类型可以转换为该接口类型。类型码为 varUnknown、varEmpty 或 varDispatch 的 Variant 变量，可以转换为 IInterface 接口类型；类型码为 varEmpty 或 varDispatch 的 Variant 变量，可以转换为 IDispatch 接口类型。

我们在示例程序 6-3 中讲过接口与类型转换的例子。以前我们接触到的多是类的类型转换包括类的向上和向下转型，现在出现在示例程序 6-3 中的却是类向继承的接口转型。为什么 TMan 类可以向接口转型，这是因为这个类与其他的类不一样。TMan 类继承自 TInterfacedObject，所以它包含有支持接口的信息。

```
TMan = class (TInterfacedObject)
```

由此可见，类向接口转型必须具备的条件是：该类或该类的基类继承自 TInterfacedObject。如果不是这样，则至少是继承自 IInterface，并提供了 IInterface 实现 _AddRef、_Release 和 QueryInterface 的方法。否则，类向接口转型会失败。

类向接口转型有时是隐式的，在程序中看起来就好像一个接口变量与实现这个接口的类是赋值相容的，例如，下面的代码是合法的：

```

type
IFoot: interface
    ['{2137BF60-AA33-11D0-A9BF-9A4537A42701}']
    .....
end;

IBall = interface
    ['{2137BF61-AA33-11D0-A9BF-9A4537A42701}']
    .....
end;

TFootball = class (TInterfacedObject, IFoot, IBall)
    .....
end;

```

```
implementation

procedure Test (FB: TFootball)
var
  F: IFoot;
begin
  F := FB; //合法,因为 FB 支持 IFoot
```

当一个类有多个继承的接口,需要在这些接口之间进行切换时,使用类型强制转换运算符 `as` 可以显式地把一个接口类型的变量强制转型为另一种接口,示例如下:

```
var
  FB: TFootBall;
  F: IFoot;
  B: IBall;
begin
  FB := TFootBall.Create;
  F := FB; //合法,因为 FB 支持 IFoot
  B := F as IBall; //把 F 转换为 IBall
```

接口转型看似神秘,其实不难理解。其核心是 Delphi 调用了 `QueryInterface` 作为对接口的 `as` 运算符的部分实现。我们可以使用 `as` 运算符来进行受检查的接口转换,这也称为接口查询。它从一个类引用转换为接口类型,或将一个接口引用转换为另一种接口类型,它基于实际的(运行时)对象类型。接口转型的格式如下:

```
object as interface
```

这里 `object` 是一个接口类型的表达式,或者是一个变体类型,或者是实现了某个接口的类的实例;`interface` 是任何一个声明了 GUID 的接口。(为了能够查询到该接口,必须有声明好的 GUID。)

如果 `object` 是 `nil`,则返回 `nil`;否则,传递 `interface` 接口的 GUID 到 `object` 的 `QueryInterface` 方法。

Delphi 调用 `QueryInterface` 来获取新的接口引用。如果 `QueryInterface` 不是返回 0,则表示 `as` 运算符将引发一个运行时错误。`SysUtils` 单元将运行时错误映射给一个 `EIntfCastError` 异常。

与接口有关的类型转换十分有用,许多 Delphi 高手喜欢使用,但却秘而不传。通过转型,在编程中我们可以轻松获得不同的接口类型和接口方法,实现一种更简洁的写法,从而使程序保持了优美和灵活的风格。

6.5 接口和多重继承

6.5.1 什么是多重继承

多重继承是 OOP 中一个很有意思而又比较费解的话题。即使许多自称掌握 OOP 的程序员也对多重继承望而生畏。其实多重继承是一个很好也很有用的概念。下面举一个多重继承的例子。

比如，一个移民到美国的中国人，与美国人结婚生下的孩子是一个中美混血儿。这个孩子既继承了中国人的血统和文化，也继承了美国人的血统和文化，是一个典型的多重继承。如果他的中国父亲教他说中文，美国母亲教他说英文，那么他就既会说中文也会说英文。用类图来表示则如图 6-3 所示。但是问题出现了，如果这个混血儿 (TAmericanChinese) 需要向上转型，是转型为中国人 (TChinese) 还是美国人 (TAmerican) 呢？

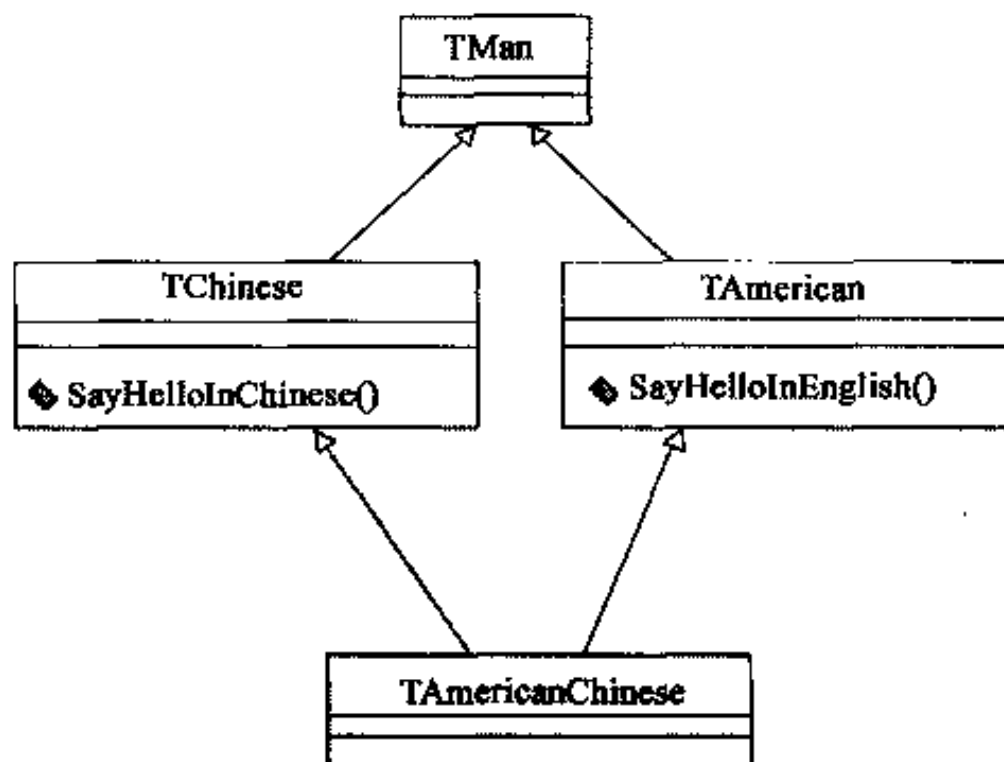


图 6-3 利用类继承实现的多重继承

类是否应该支持多重继承一直是 OOP 中存在的一个有争议的话题。虽然 C++ 的类支持多重继承，但 C++ 之后情况有所变化。比如：Java 和 C# 都不支持类的多重继承。其实 Delphi 是最早使用单根继承体系的语言，Delphi 作为重要的面向对象语言其历史可以上溯到 Object Pascal 时代。使用单根继承体系的好处是所有对象都有公共通用的接口，这使得它们最终都属于相同的类型 (Type)。特别在需要进行类型转换和类型识别时 (比如：使用多态、传递对象等情况下)，潜在的好处是显而易见的。在类似 C++ 的多重类继承设计中，虽然容易实现多重继承，但你无法保证所有对象都隶属于一个基本的类型 (Type)。在 C 模型中，转型可能不太受限制，但这在 Delphi 这样的强类型检查语言中，是无法想像的。

6.5.2 利用接口实现多重继承

那么，是不是在 Delphi 中就无法使用多重继承的概念进行面向对象编程呢？当然不是的。就以刚才的例子来说，一个中美混血儿，不可能完全继承中国人和美国人包括血统和文化在内的一切特征。虽然是多重继承，但实际上这种继承是有选择性的。图 6-3 显示了最常见的继承内容，即对语言的继承。这样一来我们可以使用接口来代替类的多重继承，如图 6-4 所示。实际上我们让混血儿和其父母一样继承了人 (TMan) 的特征，并从接口继承了父母的差异，比如使用语言上的差异。这样在子类 TAmericanChinese 中，可以从 ISpeakChinese 和 ISpeakEnglish 接口分别继承父母 SayHello 的方法，并实现之。

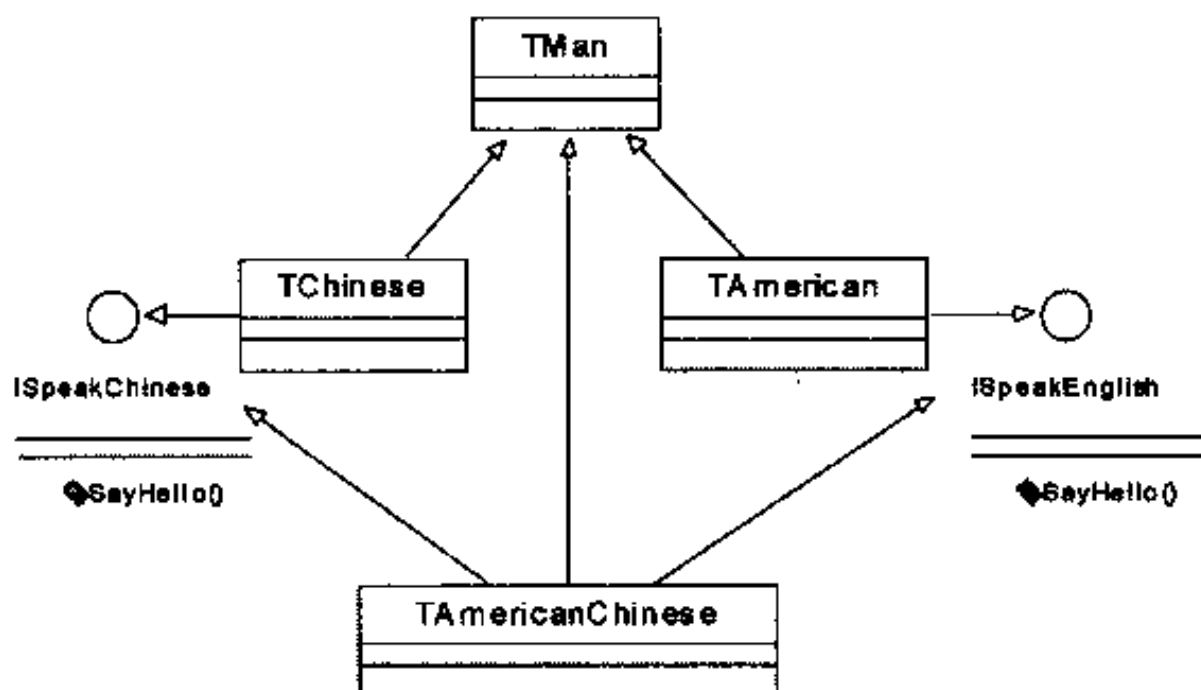


图 6-4 利用接口实现的多重继承

其实接口继承和类继承没有太大区别，只是接口继承的是定义而不是实现。图 6-4 的对应 Delphi 示例程序 6-4 的实现代码如下。

示例程序 6-4 利用接口实现的多重继承

```

unit uSayHello;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  ISpeakChinese = interface (IInterface)
    function SayHello: string;
  end;

  ISpeakEnglish = interface (IInterface)
    function SayHello: string;
  end;

  TMan = class (TInterfacedObject)
  private
    FName: string;
  public
    property Name: string read FName write FName;
  end;

  TChinese = class (TMan, ISpeakChinese)
  private
    function SayHello: string;
  end;

```



```

end;

TAmerican = class (TMan, ISpeakEnglish)
private
    function SayHello: string;
end;

TAmericanChinese = class (TMan, ISpeakChinese, ISpeakEnglish)
public
    constructor create;
    function SayHello: string;
end;

implementation

{
    * * * * * TAmerican * * * * *
}
function TAmerican.SayHello: string;
begin
    result: = 'Hello!';
end;
{
    * * * * * TChinese * * * * *
}
function TChinese.SayHello: string;
begin
    result: = '你好!';
end;

{
    * * * * * TAmericanChinese * * * * *
}
constructor TAmericanChinese.create;
begin
    name: = 'Tom Wang';
end;

function TAmericanChinese.SayHello: string;
var
    Dad: ISpeakChinese;
    Mum: ISpeakEnglish;
begin
    Dad: = TChinese.Create;
    Mum: = TAmerican.Create;
    result: = Dad.SayHello + Mum.SayHello;
end;

end.

```

```
unit frmMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class (TForm)
    Button1: TButton;
    LabeledEdit1: TLabeledEdit;
    procedure Button1Click (Sender: TObject);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation

uses uSayHello;

{$R *.dfm}

procedure TForm1.Button1Click (Sender: TObject);
var
  Tom: TAmericanChinese;
begin
  Tom := TAmericanChinese.Create;
  try
    LabeledEdit1.Text := Tom.Name;
    ShowMessage (Tom.SayHello);
  finally
    Tom.Free;
  end;
end;

end.
```

6.5.3 有侧重的多重继承

注意到图 6-4 显示的是一种对等继承，即 TAmericanChinese 通过接口继承了父母双方的 SayHello 的方法。在多重继承中，如果继承的一方有所侧重，情况又是怎样呢？图 6-5 就显示

了混血儿 (TAmericanChinese) 继承侧重于中国人 (TChinese) 一方的情况。因为他除了继承 ISpeakChinese 和 ISpeakEnglish 接口, 还继承自 TChinese 类, 以获得中国人 (TChinese) 的所有特征。

图 6-6 作为一个用 ModelMaker 设计的不对等继承的例子, 其 Delphi 实现代码如示例程序 6-5 所示。

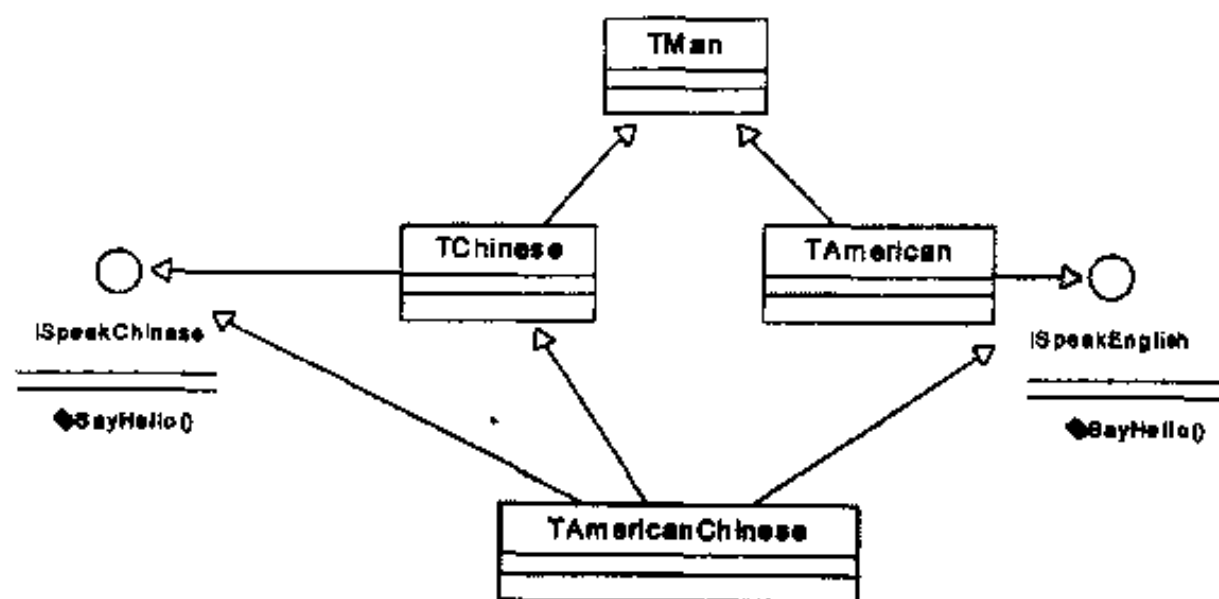


图 6-5 不对等的多重继承

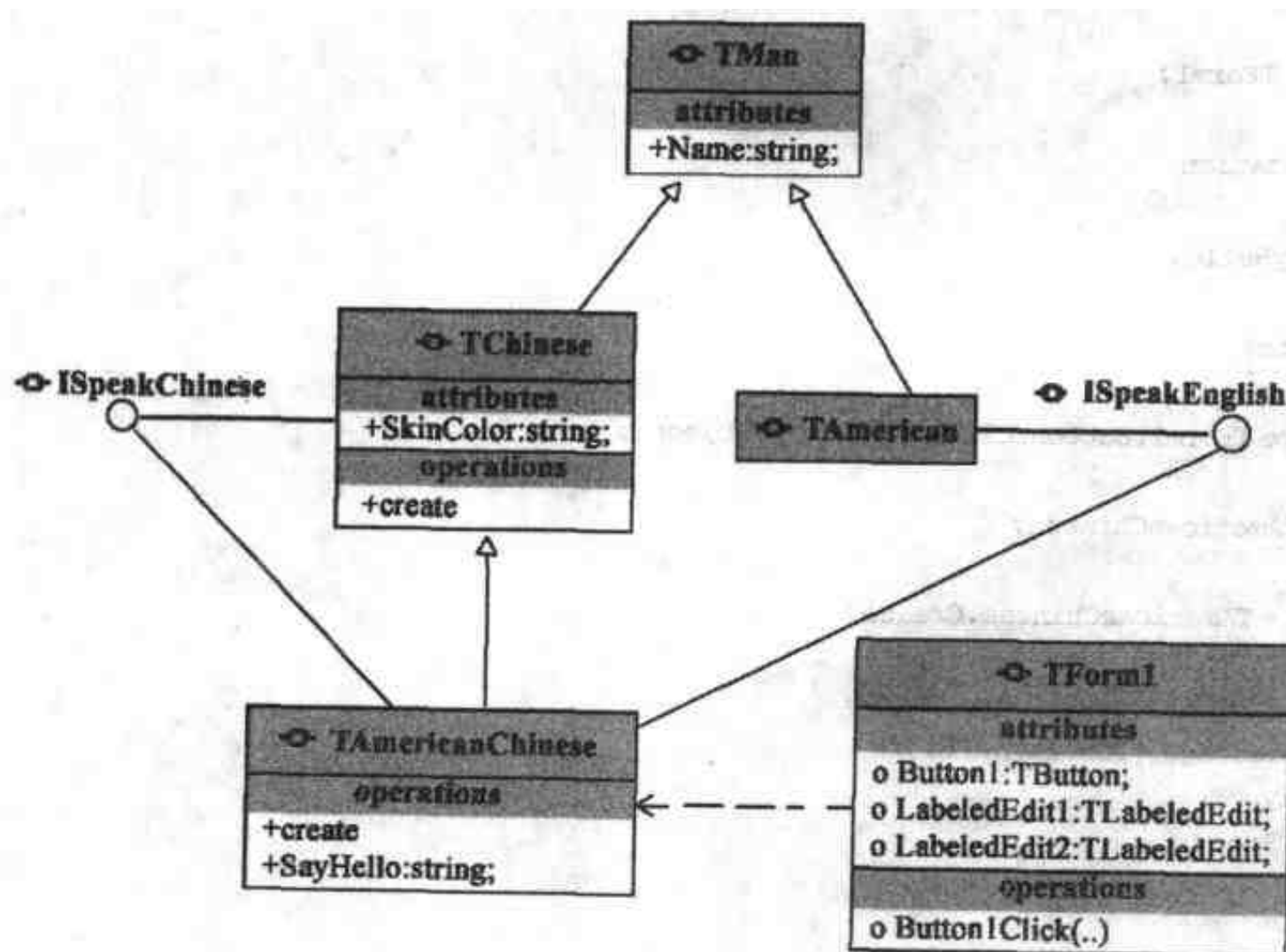


图 6-6 用 ModelMaker 设计的一个不对等继承的例子

示例程序 6-5 不对等的多重继承

```
unit uSayHello;
```

```
interface
```

```

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  ISpeakChinese = interface (IInterface)
    function SayHello: string;
  end;

  ISpeakEnglish = interface (IInterface)
    function SayHello: string;
  end;

  TMan = class (TInterfacedObject)
  private
    FName: string;
  public
    property Name: string read FName write FName;
  end;

  TChinese = class (TMan, ISpeakChinese)
  private
    FSkinColor: string;
    function SayHello: string;
  public
    constructor create;
    property SkinColor: string read FSkinColor write FSkinColor;
  end;

  TAmerican = class (TMan, ISpeakEnglish)
  private
    function SayHello: string;
  end;

//注意,这里 TAmericanChinese 的声明和前面的示例程序不一样
  TAmericanChinese = class (TChinese, ISpeakChinese, ISpeakEnglish)
  public
    constructor create;
    function SayHello: string;
  end;

implementation

{
  * * * * * TAmerican
}

function TAmerican.SayHello: string;
begin
  result: = 'Hello! ';

```

```

end;

{
  * * * * * TChinese
}
constructor TChinese.create;
begin
  skincolor: = '黄色';
end;

function TChinese.SayHello: string;
begin
  result: = '你好!';
end;

{
  * * * * * TAmericanChinese
}
constructor TAmericanChinese.create;
begin
  name: = 'Tom Wang';
  Inherited;
end;

function TAmericanChinese.SayHello: string;
var
  Dad: ISpeakChinese;
  Mum: ISpeakEnglish;
begin
  Dad: = TChinese.Create;
  Mum: = TAmerican.Create;
  result: = Dad.SayHello + Mum.SayHello;
end;

end.

unit frmMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class (TForm)
    Button1: TButton;
    LabeledEdit1: TLabeledEdit;
    LabeledEdit2: TLabeledEdit;
    procedure Button1Click (Sender: TObject);
  end;

```

```
private
  {Private declarations }
public
  {Public declarations }
end;

var
  Form1: TForm1;

implementation

uses uSayHello;

{$R *.dfm}

procedure TForm1.Button1Click (Sender: TObject);
var
  Tom: TAmericanChinese;
begin
  Tom := TAmericanChinese.Create;
  try
    LabeledEdit1.text := Tom.Name;
    LabeledEdit2.text := Tom.SkinColor;
    Showmessage (Tom.sayhello);
  finally
    Tom.Free;
  end;
end;

end.
```

注意到示例程序 6-5 中的 TAmericanChinese 是这样声明的:

```
TAmericanChinese = class (TChinese, ISpeakChinese, ISpeakEnglish)
```

这显然可以看出来其继承自 TChinese、ISpeakChinese 和 ISpeakEnglish。但接口往往是提供给类的非直接继承对象的,其目的是将类继承与类型(接口是一种类型)继承分离开来。在有侧重的多重继承问题上,如果我们对继承的类感兴趣,比如这里我们需要 TAmericanChinese 继承 TChinese 全部属性和方法,就可以采用类继承辅以接口继承的方法,如图 6-7 所示。新的 TAmericanChinese 声明改为:

```
TAmericanChinese = class (TChinese, ISpeakEnglish)
```

显然 ISpeakChinese 的 SayHello 方法已经包含在 TChinese 的实现中了。如果我们新增 TomSayHello 方法,就不需要创建原来的 ISpeakChinese 接口,并调用其 SayHello 方法。因为这里直接使用的 SayHello 方法是 TAmericanChinese 继承自其基类 TChinese 的。

```
function TAmericanChinese.TomSayHello: string;
var
```

```

//Dad: ISpeakChinese;
Mum: ISpeakEnglish;
begin
//Dad: = TChinese.Create;
Mum: = TAmerican.Create;
result: = SayHello + Mum.SayHello;
// 这里直接使用的 SayHello 方法是 TAmericanChinese 继承自其基类 TChinese 的
end;

```

既然可以声明 `TAmericanChinese = class (TChinese, ISpeakEnglish)`, 那么原来在 `TAmericanChinese = class (TChinese, ISpeakChinese, ISpeakEnglish)` 声明中继承 `ISpeakChinese` 接口岂不是多此一举? 未必! 从图 6-6 和图 6-7 的 UML 设计图看, 前者强调的是接口继承而后者侧重的是类继承。对于前者, 其设计思路着眼于封装, 示例程序 6-5 实际上封装了 `TChinese` 的 `SayHello` 方法 (private 方法), 有意让用户使用接口来访问 `TChinese`。而后者只是侧重于直接使用基类的方法, 享用类继承的方便。

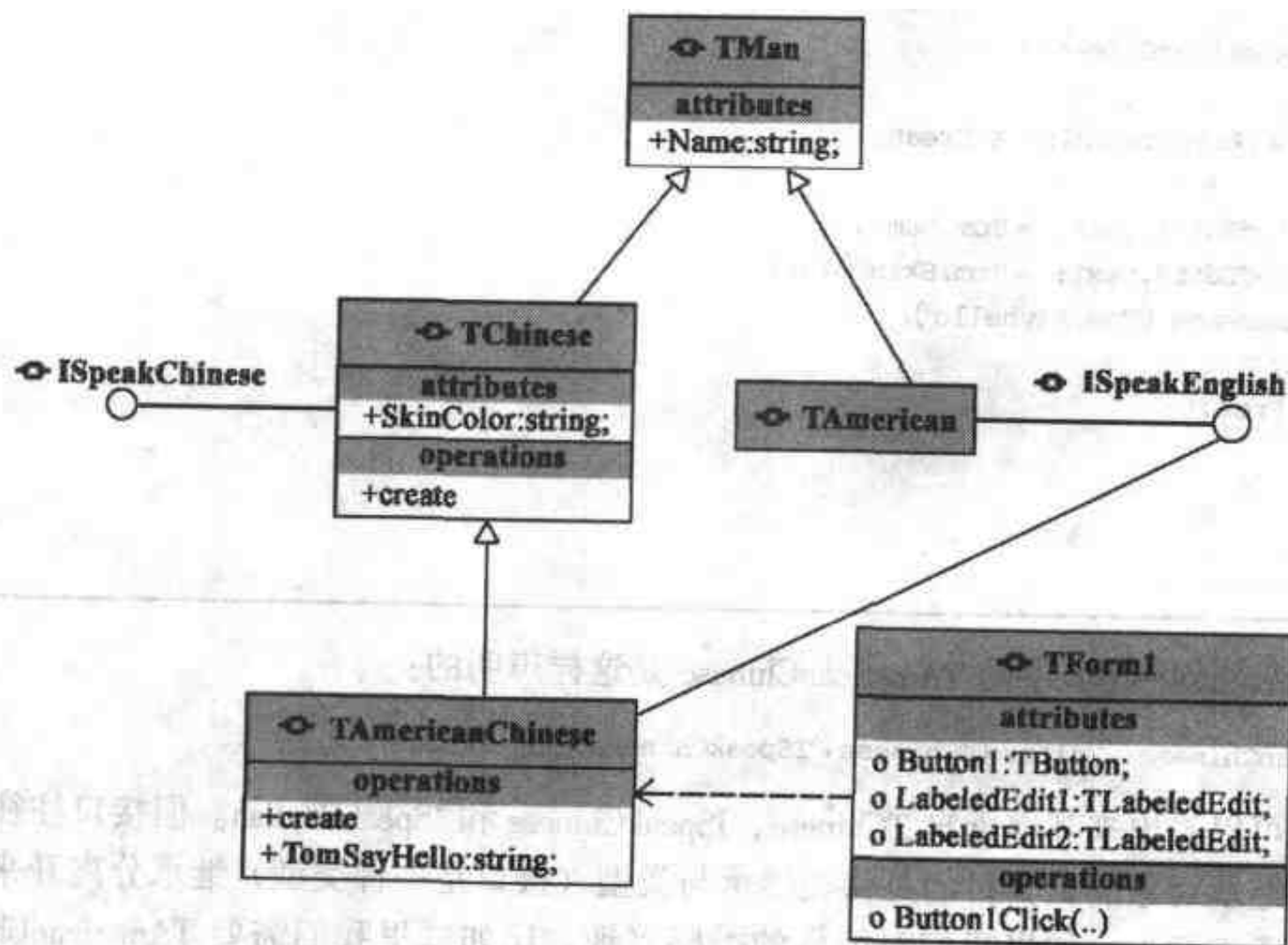


图 6-7 如果多重继承有侧重, 也可以采用类继承辅以接口继承的方法

可能会有读者不明白, 既然 `TChinese` 的 `SayHello` 方法是 private 方法, 为什么在 `TAmericanChinese` 中不通过接口仍然可以利用类继承来调用? 那是因为在同一个 Delphi 单元文件中, 所有的类都可以访问同单元的其他类的私有成员。不信的话, 你可把 `TAmericanChinese` 和 `TChinese` 分开在不同的单元文件中看看, private 方法可不是那么随便可以访问的。

6.5.4 多重继承的深入讨论

使用接口来实现多重继承最大的好处就是灵活。比如我们要为 `TAmericanChinese` 增加新的

继承基类，则只需为 TAmericanChinese 增加新的接口，如图 6-8 所示。类可以实现任意多的接口，如图 6-9 所示。派生类可以继承的接口数目不受限制，而且每一个都可以是独立的类型，并都可以成为向上转型的目标。利用继承，我们可以轻易地将新的方法加之接口或派生类中；也可以通过接口继承接口，将多个接口合并成一个新的接口。

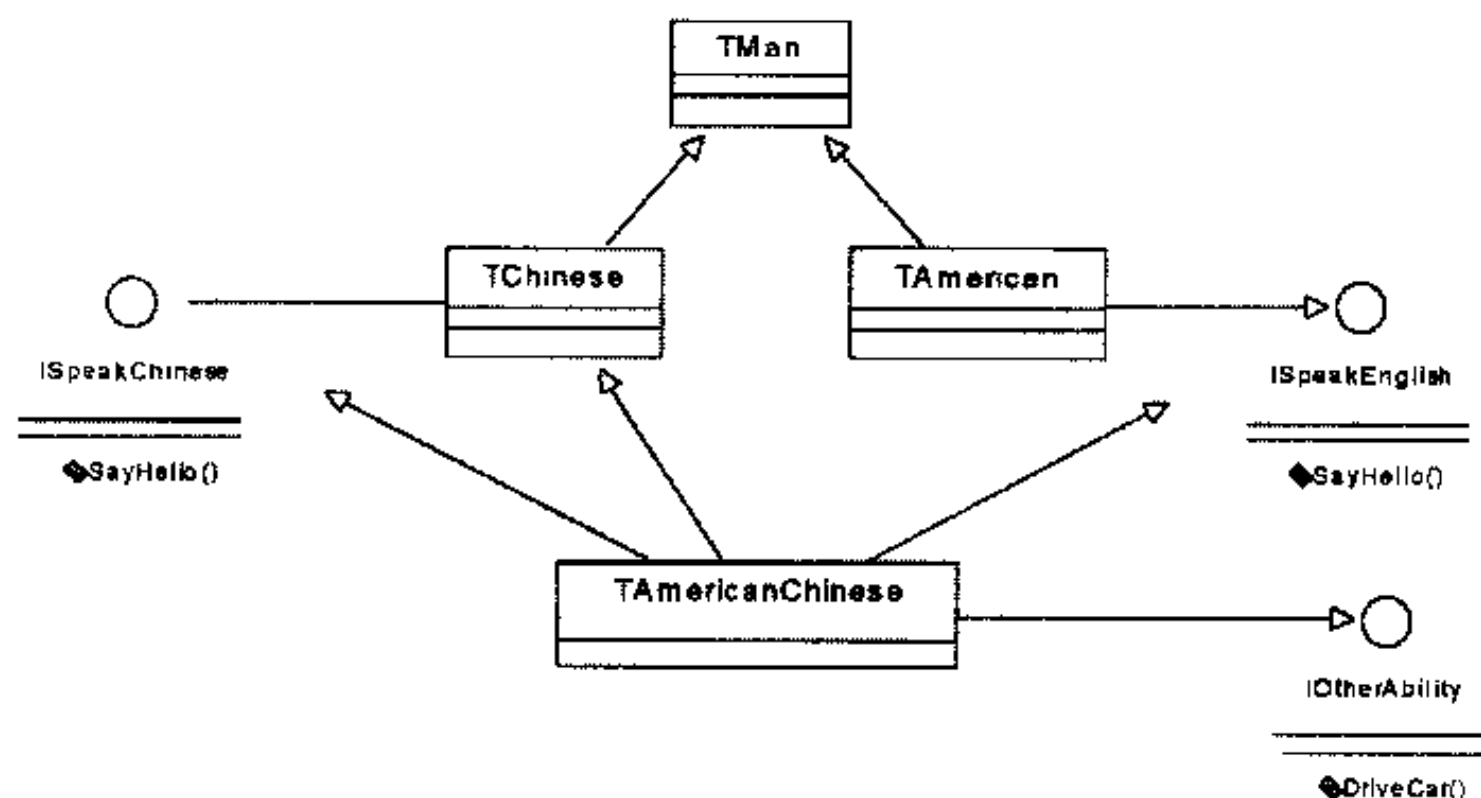


图 6-8 多重继承的扩展

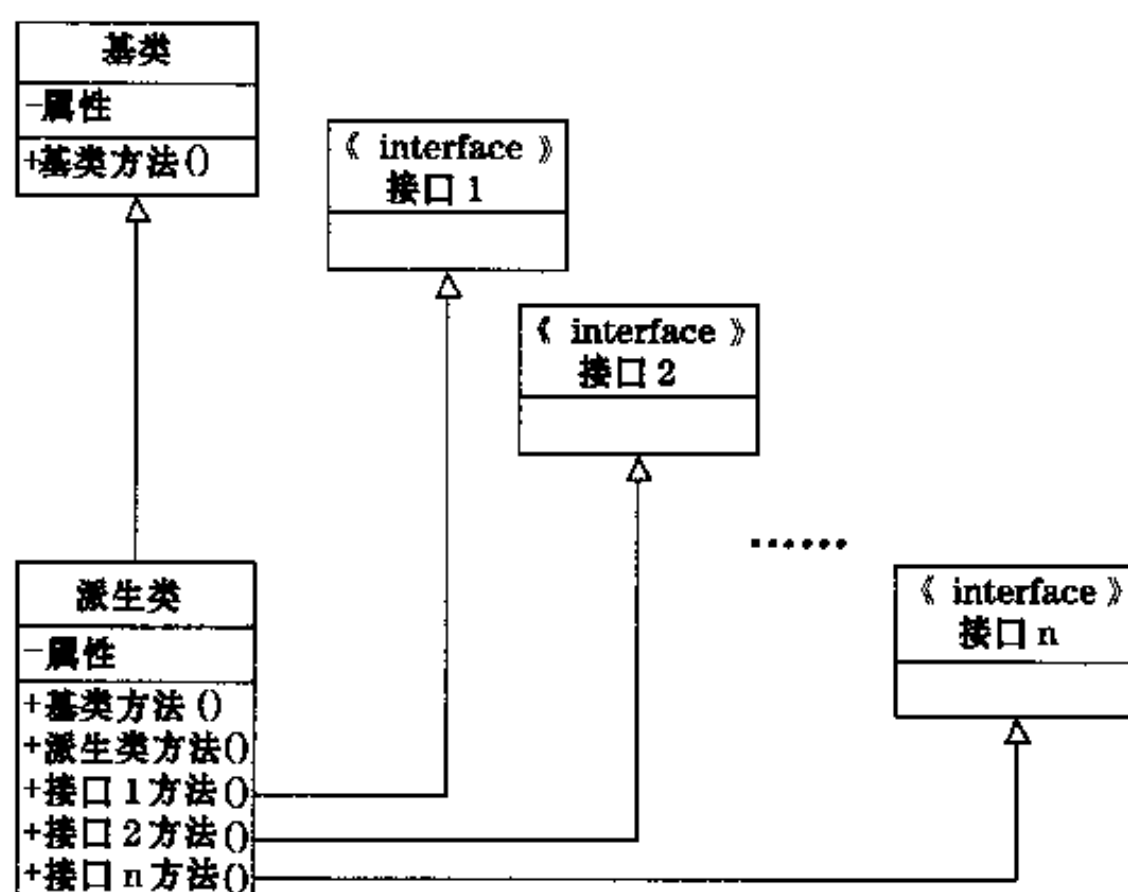


图 6-9 类可以实现任意多的接口

接口自身也是可以继承的。我们可以利用接口继承的特性，继承或覆盖祖先接口的方法。图 6-10 显示了利用接口继承来实现多重继承的设计方法。相对应的示例程序 6-6 的实现代码如下所示。

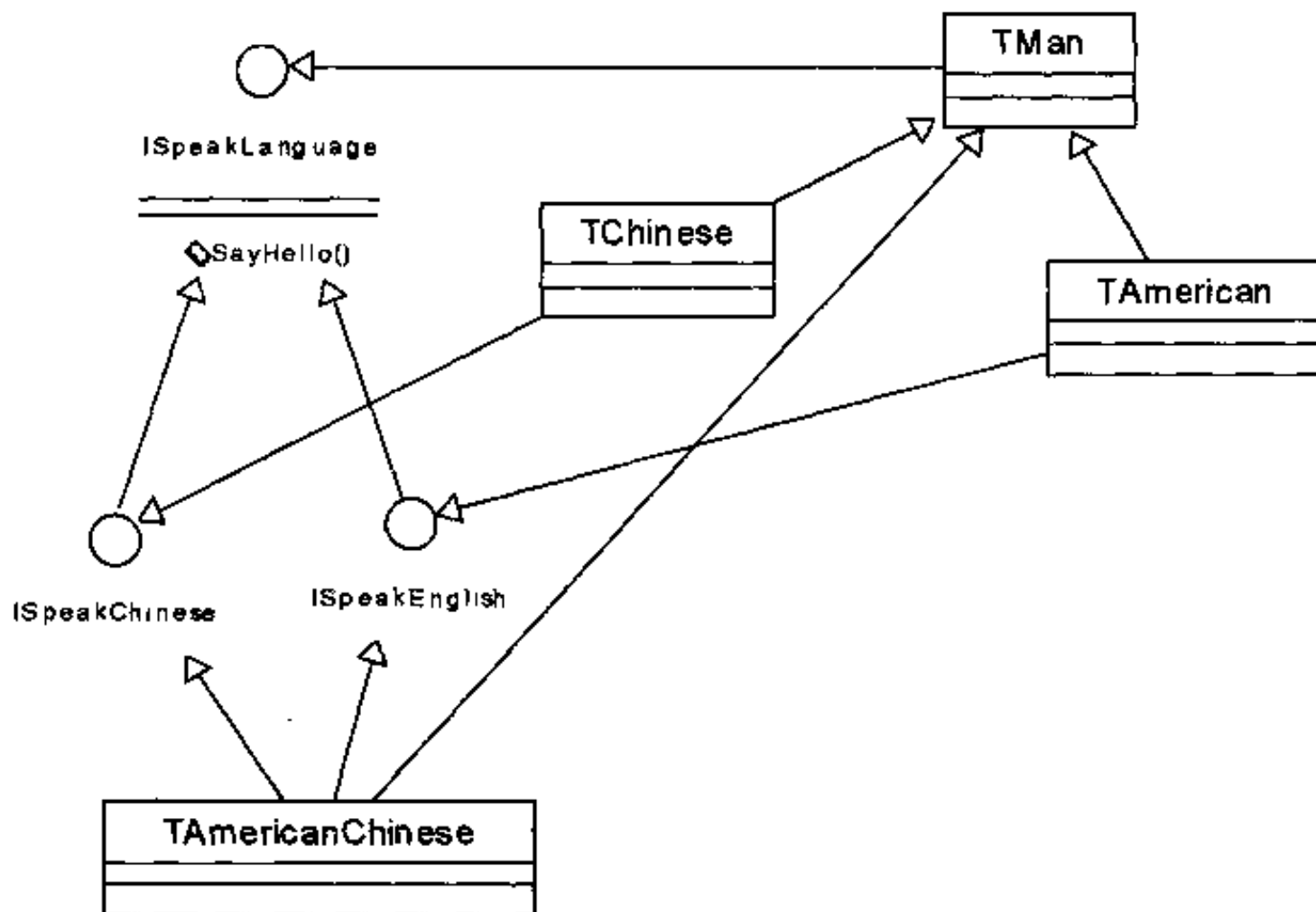


图 6-10 接口继承的多重继承

示例程序 6-6 接口继承的多重继承

```

unit uSayHello;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  ISpeakLanguage = interface (IInterface)
    function SayHello: string;
  end;

  ISpeakChinese = interface (ISpeakLanguage)
    //function SayHello: string;
  end;

  ISpeakEnglish = interface (ISpeakLanguage)
    //function SayHello: string;
  end;

  TMan = class (TInterfacedObject)
  private
    FName: string;
  public
    property Name: string read FName write FName;
  end;

```

```

TChinese = class (TMan, ISpeakChinese)
private
    function SayHello: string;
end;

TAmerican = class (TMan, ISpeakEnglish)
private
    function SayHello: string;
end;

TAmericanChinese = class (TMan, ISpeakChinese, ISpeakEnglish)
public
    constructor create;
    function SayHello: string;
end;

implementation

{
    * * * * * TAmerican
}
function TAmerican.SayHello: string;
begin
    result: = 'Hello! ';
end;

{
    * * * * * TChinese
}
function TChinese.SayHello: string;
begin
    result: = '你好! ';
end;

{
    * * * * * TAmericanChinese
}
constructor TAmericanChinese.create;
begin
    name: = 'Tom Wang';
end;

function TAmericanChinese.SayHello: string;
var
    Dad: ISpeakChinese;
    Mum: ISpeakEnglish;
begin
    Dad: = TChinese.Create;
    Mum: = TAmerican.Create;
    result: = Dad.SayHello + Mum.SayHello;
end;

end.

```

在示例程序 6-6 中, 考虑到 `ISpeakChinese` 和 `ISpeakEnglish` 都有共同的 `SayHello` 方法, 利用其共同的祖先接口 `ISpeakLanguage`, 我们可以将 `SayHello` 方法分别继承到 `ISpeakChinese` 和 `ISpeakEnglish`, 而不必在这两个接口中重新声明 `SayHello` 方法了。接口继承与类继承不同, 接口继承仅仅是为了输入方便, 不需要重新输入很多的方法声明。当类实现接口时, 并不意味着类自动实现了其祖先接口, 类仅仅实现那些列入其声明 (以及其祖先类声明) 之中的接口。

6.6 接口和面向对象编程

接口语言可以看做包含抽象函数的最终抽象类。甚至从句法上讲, 接口与只包含抽象函数的抽象类相似。

一个类可以实现一个接口, 如此一来, 类就要实现接口的抽象函数成员, 与派生类保证要实现它的基类的抽象函数一样。所以接口是通过动态绑定函数调用抽象基类的替代方式。例如在前面的示例程序 6-1 和示例程序 6-2 中, 我们可以用一个等价的 `IGreetable` 接口来替换 `TMan` 类, 此接口也包含一个抽象的 `SayHello` 方法。`TChinese`、`TAmerican`、`TFrench` 等派生类也将实现 `IGreetable` 接口。这不会影响动态绑定调用 `SayHello` 函数的能力。不过这个例子还不能演示和说明接口的威力和妙用。

即使接口和抽象类在句法和语义上密切相关, 但它们仍有一个重要的区别: 接口只能包含抽象方法, 而抽象类除了包含抽象方法外, 还可以包含其他数据成员和非抽象方法。另外, 一个类最多只有一个基类, 但可以有多个接口。

一个接口可以自身实现一个或多个接口, 从而继承这些接口的方法, 形成与类层次一样的接口层次。

在前面的内容中, 普通类形成了健全的分类层次, 其中每一对基类和派生类表示一个“is-a”关系。这些层次提供了极大的好处, 但也从一定程度上限制了多态的使用。原因是: 在一个类层次中要充分利用多态, 必须具备一组相同的祖先类。此祖先类声明了接口 (通常以抽象方法形式) 以及派生类中单个实现过程间的通信协议。如果我们想要划分成同一组的类, 但不具有相同的祖先类怎么办呢? 如果它们分布于整个程序的多个地方, 甚至位于多个单独的类层次中, 又该如何处理呢?

假如我们重新考虑“来自世界的问候”的那个例子, 将类层次由一个 `TMan` 增加到 `TUnkown`、`TMachine` 等数个, 如图 6-11 所示。其中, 我们希望机器人和外星人都能像人类一样进行问候, 都有自己的 `SayHello` 方法 (尽管可能问候的方式不同)。但我们无法将它们归纳到一个共同的祖先类。如果硬要用一个抽象类放置到 `TMan`、`TUnkown` 和 `TMachine` 的上层, 并包含一个抽象方法 `SayHello` 的话, 虽然可以工作, 但却扰乱了类的层次关系, 最终导致设计概念上的不伦不类关系。

最理想的解决方式是创建一个接口, 取名为 `IGreetable`, 用它来包含抽象方法 `SayHello`, 并让 3 个类实现这一接口, 如图 6-11 所示。该接口声明如下:

```
IGreetable = interface
    ['{D91DDE09-0FC4-4FE9-AE0D-9877E2F73BF6}']
```

```
function SayHello: string;
end;
```

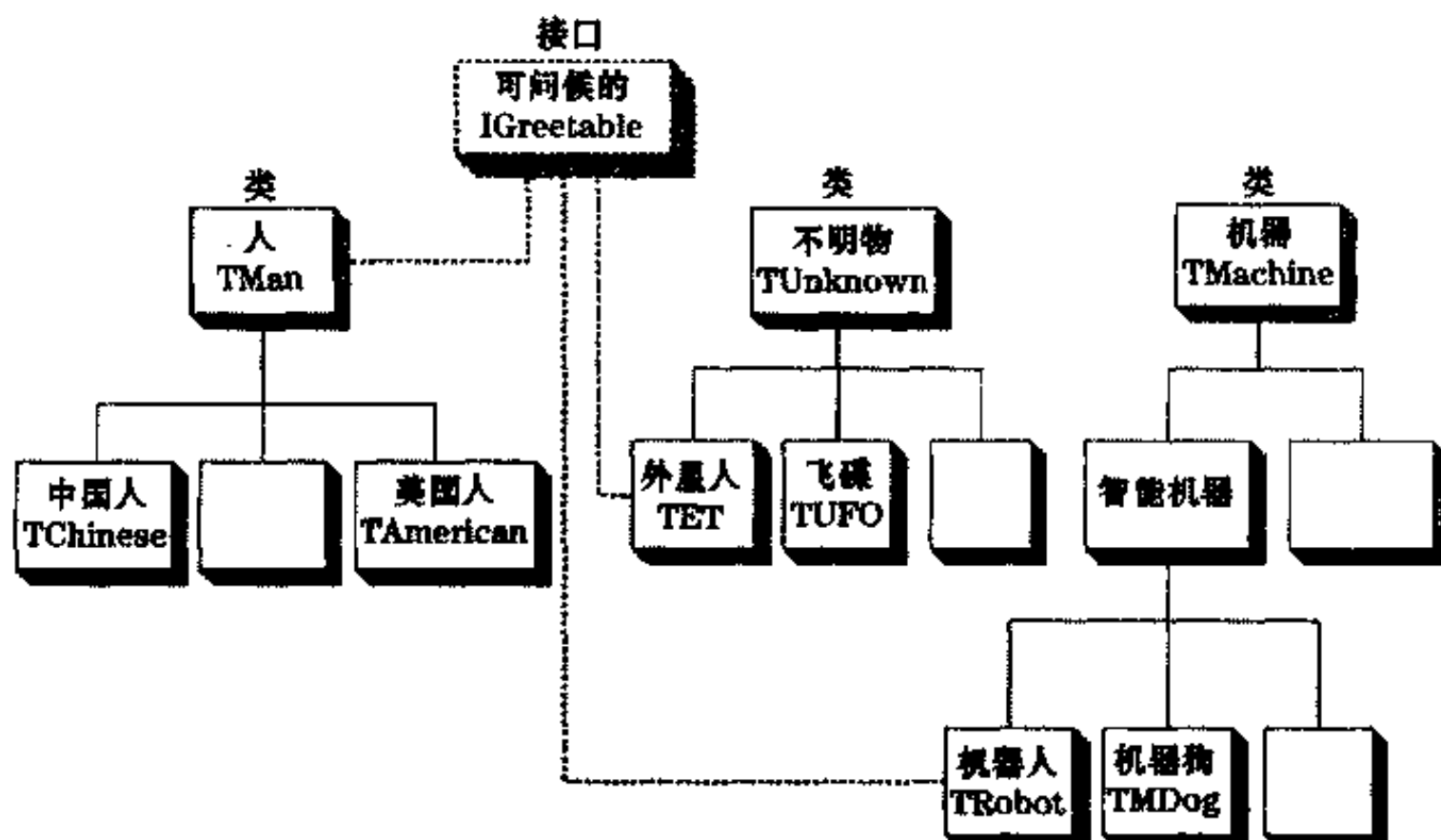


图 6-11 接口让 3 个无关的类实现相同的抽象方法

在前面“来自世界的问候”的那个例子中，我们已经初步体验到多态的作用。

在示例程序 6-7 中，TMan、TET 和 TRobot 三个没有血缘关系的类都继承自 IGreetable，它们实现 SayHello 方法的内容也不一样。有意思的是，在 TManSayHello 中它们仍然可以用一个 sayhello (greeting: IGreetable) 来实现多态，如示例程序 6-8 和图 6-12 所示。

示例程序 6-7 扩充新类 TET 和 TRobot 后的 uSayHello

```
unit uSayHello;

interface

type

  IGreetable = interface
    ['{D91DDDE09-0FC4-4FE9-AE0D-9877E2F73BF6}']
    function SayHello: PChar;
  end;

  TMan = class (TInterfacedObject, IGreetable)
  public
    Language: string;
    Married: Boolean;
    Name: string;
    SkinColor: string;
    constructor create; virtual;
    function SayHello: PChar; virtual; abstract;
  end;
```

```
TChinese = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TAmerican = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TFrench = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TKorean = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TET = class (TInterfacedObject, IGreetable)
private
    function SayHello: PChar;
end;

TRobot = class (TInterfacedObject, IGreetable)
private
    function SayHello: PChar;
end;

implementation

constructor TMan.create;
begin
    Name: = '张三';
    Language: = '中文';
    SkinColor: = '黄色';
end;

constructor TChinese.create;
begin
    inherited;
end;

constructor TAmerican.create;
begin
```

```
Name: = 'Lee';
Language: = '英文';
SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
    Name: = '苏菲';
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;

function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

function TET.SayHello;
begin
    Result: = 'ET.bmp';
end;

function TRobot.SayHello;
begin
    Result: = 'Robot.bmp';
end;

end.
```

 示例程序 6-8 扩充新类 TET 和 TRobot 后的 ufrmSayHello

```

unit ufrmSayHello;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, uSayHello;

type
  TfrmSayHello = class (TForm)
    btnUSA: TButton;
    btnKorean: TButton;
    btnCN: TButton;
    btnFrench: TButton;
    Image1: TImage;
    btnET: TButton;
    btnRobot: TButton;
    procedure btnUSAClick (Sender: TObject);
    procedure btnCNClick (Sender: TObject);
    procedure btnFrenchClick (Sender: TObject);
    procedure btnKoreanClick (Sender: TObject);
    procedure btnETClick (Sender: TObject);
    procedure btnRobotClick (Sender: TObject);
  private
    procedure sayhello (greeting: IGreetable);
  public
    {Public declarations}
  end;

var
  frmSayHello: TfrmSayHello;

implementation

{$R *.dfm}
procedure TfrmSayHello.sayhello (greeting: IGreetable);
begin
  image1.Picture.LoadFromFile (greeting.sayhello);
end;

procedure TfrmSayHello.btnUSAClick (Sender: TObject);
begin
  sayhello (TAmerican.create);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject);
begin
  sayhello (TChinese.create);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject);

```

```
begin
    sayhello (TFrench.create);
end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject);
begin
    sayhello (TKorean.create);
end;

procedure TfrmSayHello.btnETClick (Sender: TObject);
begin
    sayhello (TET.create);
end;

procedure TfrmSayHello.btnRobotClick (Sender: TObject);
begin
    sayhello (TRobot.create);
end;

end.
```



图 6-12 使用 IGreetable 接口后，外星人也能 SayHello

在示例程序 6-7 中，我们可以看出 IGreetable 接口的 SayHello 方法在 TMan、TET 和 TRobot 中实现方法也有区别。TET 和 TRobot 中是直接实现 SayHello 方法的。而 TMan 并没有这样做，它只是将 SayHello 方法作为抽象方法留给派生类去实现。

请注意，继承关系常常描述成“is-a”（是一个）关系，例如，TChinese “is-a” TMan。在 Delphi 的 is 运算符中可以看到相同的概念，它可以用于测试一个 AMan 变量是否“is-a” TChinese。但是，在很多复杂的情况下，简单的“is-a”关系会被打破。比如，中国人“是—

个”人，但机器人并不“是一个”人，但这并不意味着机器人不能用自己的方式向你问候。可我们决不会让机器人从 TMan 继承下来。类继承会迫使派生类存储所有在基类中声明了的数据成员，在这种情况下，将导致派生类并不需要那样的信息，比如：机器人不会需要肤色（没有皮肤的金属机器人）。尽管如此，类继承仍是代码重用的一个有效工具。派生类能轻易地继承基类的数据成员、方法和属性，从而避免重复实现普通的方法。比如，TChinese 需要继承 TMan 的 Language、Married、Name、SkinColor 数据成员和 create、SayHello 方法。

通过面向对象的分析，我们可以将中国人和机器人泛化成能问候（something greetable）的某种类型对象，并抽象出 IGreetable 接口。这样使用接口还有一个最大的好处就是，将类型继承与类继承分离开来。在一个强类型化的语言（如 Delphi）中，编译器将类作为类型对待，因此类继承就变成类型继承了。但是我们要明白：类是一种类型，类的继承和类型的继承并不完全一样。比如，接口的继承就是一种类型的继承，它和类的继承决不同。前面我们就是将类型继承（中国人和机器人都是一种能问候的类型）与类继承（类 TChinese 继承类 TMan 的数据成员和方法）分离开来。这就是说，在类型继承时使用接口，以提高程序可扩展性；而类继承仅在需要用到的地方使用，继承了的数据成员和方法可提高程序的可重用性。下面举例说明。

为了进一步演示类继承和类型继承的差别和类型继承与类继承分离的好处，我们修改一下前面“来自世界的问候”的那个例子，如示例程序 6-9、6-10 所示。

示例程序 6-9 进一步修改的 uSayHello

```
unit uSayHello;

interface

type

    IGreetable = interface
        ['{D91DDE09-0FC4-4FE9-AE0D-9877E2F73BF6}']
        function SayHello: PChar;
    end;

    TMan = class (TInterfacedObject, IGreetable)
    public
        Language: string;
        Married: Boolean;
        Name: string;
        SkinColor: string;
        constructor create; virtual;
        function SayHello: PChar; virtual; abstract;
    end;

    TChinese = class (TMan)
    public
        constructor create; override;
    private
        function SayHello: PChar; override;
    end;
```

```
TAmerican = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TFrench = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TKorean = class (TMan)
public
    constructor create; override;
private
    function SayHello: PChar; override;
end;

TET = class (TInterfacedObject, IGreetable)
private
    function SayHello: PChar;
end;

TRobot = class (TInterfacedObject, IGreetable)
private
    function SayHello: PChar;
end;

implementation

constructor TMan.create;
begin
    Name: = '张三';
    Language: = '中文';
    SkinColor: = '黄色';
end;

constructor TChinese.create;
begin
    inherited;
end;

constructor TAmerican.create;
begin
    Name: = 'Lee';
    Language: = '英文';
    SkinColor: = '黑色';
end;

constructor TFrench.create;
begin
```

```

    Name: = '苏菲';
    Language: = '法文';
    SkinColor: = '白色';
end;

constructor TKorean.create;
begin
    Name: = '金知中';
    Language: = '韩文';
    SkinColor: = '黄色';
end;

function TChinese.SayHello;
begin
    Result: = 'chinese.bmp';
end;

function TAmerican.SayHello;
begin
    Result: = 'American.bmp';
end;

function TFrench.SayHello;
begin
    Result: = 'French.bmp';
end;

function TKorean.SayHello;
begin
    Result: = 'Korean.bmp';
end;

function TET.SayHello;
begin
    Result: = '外星人#$$^&* (@@)';
end;

function TRobot.SayHello;
begin
    Result: = '机器人 011100001111000011111';
end;

end.

```

示例程序 6-10 进一步修改的 ufrmSayHello

```

unit ufrmSayHello;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, ExtCtrls, uSayHello;

```

```

type
  TfrmSayHello = class (TForm)
    GroupBox1: TGroupBox;
    edtName: TLabelledEdit;
    edtSkinColor: TLabelledEdit;
    edtLanguage: TLabelledEdit;
    btnUSA: TButton;
    btnKorean: TButton;
    btnCN: TButton;
    btnFrench: TButton;
    Image1: TImage;
    btnET: TButton;
    btnRobot: TButton;
    procedure btnUSAClick (Sender: TObject);
    procedure btnCNClick (Sender: TObject);
    procedure btnFrenchClick (Sender: TObject);
    procedure btnKoreanClick (Sender: TObject);
    procedure btnETClick (Sender: TObject);
    procedure btnRobotClick (Sender: TObject);
  private
    procedure sayhello (AMan: TMan); overload;
    procedure sayhello (greeting: IGreetable); overload;
  public
    {Public declarations }
  end;

```

```

var
  frmSayHello: TfrmSayHello;

```

```

implementation

```

```

{$R *.dfm}

```

//这里的实现方法和原来（示例程序 5-6）的一样,没有改动

```

procedure TfrmSayHello.sayhello (AMan: TMan);
begin
  edtName.Text: = AMan.Name;
  edtLanguage.Text: = AMan.Language;
  edtSkinColor.Text: = AMan.SkinColor;
  image1.Picture.LoadFromFile (AMan.sayHello);
end;

```

//这里新增来自非人类的另类问候实现方法。

//和示例程序 5-6 比较,这是惟一的改动。

//通过方法重载,使程序的改动降低到最少。

```

procedure TfrmSayHello.sayhello (greeting: IGreetable);
begin
  edtName.Text: = copy (greeting.sayhello, 1, 6);
  edtLanguage.Text: = copy (greeting.sayhello, 7, 4);
  edtSkinColor.Text: = copy (greeting.sayhello, 11, 6);
  application.MessageBox (greeting.sayhello,
    '问候', MB _ ICONINFORMATION + MB _ OK );
end;

```

```
procedure TfrmSayHello.btnUSAClick (Sender: TObject);
begin
    sayhello (TAmerican.create);
end;

procedure TfrmSayHello.btnCNClick (Sender: TObject);
begin
    sayhello (TChinese.create);
end;

procedure TfrmSayHello.btnFrenchClick (Sender: TObject);
begin
    sayhello (TFrench.create);
end;

procedure TfrmSayHello.btnKoreanClick (Sender: TObject);
begin
    sayhello (TKorean.create);
end;

procedure TfrmSayHello.btnETClick (Sender: TObject);
begin
    sayhello (TET.create);
end;

procedure TfrmSayHello.btnRobotClick (Sender: TObject);
begin
    sayhello (TRobot.create);
end;

end.
```

大家可能会注意到我在示例程序 6-10 中分别重载了 sayhello 方法，于是可以在 TfrmSayHello（实际上是界面类，通常是位于表示层中）中将人类的和非人类的 sayhello 方法分别实现，不过实现的代码完全不一样。

```
procedure sayhello (AMan: TMan); overload;
procedure sayhello (greeting: IGreetable); overload;
```

虽然按钮单击事件中的写法都是 Txxx.create，但是作为 TMan 的派生类都能通过重载找到适合自己的 sayhello (AMan: TMan) 方法，并实现自己的显示内容；TET 和 TRobot 也能通过重载找到适合自己的 sayhello (greeting: IGreetable) 方法，实现与 TMan 的派生类完全不同的显示内容。值得注意的是，示例程序 6-10 中的 TfrmSayHello.sayhello 方法已经不仅仅像示例程序 6-8 中那样仅有一条代码：

```
image1.Picture.LoadFromFile (greeting.sayhello);
```

这意味着类型继承与类继承分离后，我们可以享受到“鱼与熊掌”兼得的好处：既通过接口实现多态的灵活性，又通过继承获得代码的重用性。实际上运行该程序，我们也能够立即观察到它既保留了原有的功能（来自世界各国的问候），又扩展了新的功能（来自非人类的另类问候）。

这个例子巧妙地用到了覆盖、重载、继承、多态、抽象方法和抽象类、虚方法、接口、类的类型转换（类的向上转型，类向继承的接口转型）等众多概念，如果能够深入理解掌握，必将会使你在面向对象编程方面功力大增。因此，值得读者好好研习和体会。

6.7 接口的其他用法探索

本章已经介绍了有关接口与面向对象编程的不少知识。不过除了深入学习和不断实践外，要提高编程水平还需要多动脑筋，开拓创新。为了进一步启发思路，下面我介绍接口的一种另类用法。

我们知道对象是有生命期的，创建对象需要占用一定的资源，当对象的生命期结束后，我们需要记住释放这些资源，否则内存中会存在大量的垃圾。Delphi 没有提供垃圾自动回收功能，不过 Delphi 的接口可以通过 `_AddRef` 和 `_Release` 方法来增减实例计数器，从而实现对象实例的管理，安全销毁不再需要的对象。

澳大利亚的 Malcolm Groves 曾经使用 Delphi 的接口技术写过一个简单的垃圾自动回收程序，如示例程序 6-11 所示。

这里包括了 `mtReaper` 和 `frmMain` 两个单元，其中 `mtReaper` 单元定义了一个名为 `ImtReaper` 的接口，其中没有任何方法，仅仅是为了利用接口的实例引用计数器管理功能。这种类型的接口也称为标记（Tag）接口，它的存在表明类能以一种特定的方式被使用。`mtReaper` 单元还定义了一个名为 `TmtReaper` 的类，它继承于 `TInterfacedObject` 类，并实现了 `ImtReaper` 接口。我们知道 `TInterfacedObject` 类已经实现了在 `IInterface`（在 Delphi 6 之前是 `IUnknown`）中的一些方法，包括：`QueryInterface`、`_AddRef` 和 `_Release` 方法，这正是我们所需要的。最后在 `mtReaper` 单元中还使用了一个构造函数、一个析构函数以及一个类型为 `TObject` 的私有变量 `FObject`。在构造函数中，我们将一个对象的引用作为它的参数，并且将该引用保存在私有变量 `FObject` 中。`FObject` 随后将对象实例在析构函数中销毁。

示例程序 6-11 一个简单的垃圾自动回收程序

```
unit mtReaper;  
  
interface  
type  
  ImtReaper = interface  
    ['{F3E97960-3F35-11D7-B847-001060806215}']  
  end;  
  
  TmtReaper = class (TInterfacedObject, ImtReaper)  
  private  
    FObject: TObject;  
  public  
    constructor Create (AObject: TObject);  
    destructor Destroy; override;  
  end;  
  
implementation
```



```

uses SysUtils;

constructor TmtReaper.create (AObject: TObject);
begin
    FObject := AObject;
end;

destructor TmtReaper.Destroy;
begin
    FreeAndNil (FObject);
    Inherited;
end;

end.

unit frmMain;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
    TNoisyDeath = class (TObject)
    public
        destructor Destroy; override;
    end;

    TForm1 = class (TForm)
        Button1: TButton;
        Button2: TButton;
        procedure Button1Click (Sender: TObject);
        procedure Button2Click (Sender: TObject);
    private
        procedure WaitAWhile;
    public
        {Public declarations}
    end;

var
    Form1: TForm1;

implementation

uses mtReaper;
{$R *.dfm}

destructor TNoisyDeath.Destroy;
begin
    showMessage ('对象销毁了! ');
    inherited;
end.

```

```
end;

procedure TForm1.WaitAWhile;
var i: Integer;
begin
  for i := 0 to 5000 do
  begin
    caption := IntToStr(i);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
var NoisyDeath: TNoisyDeath;
begin
  NoisyDeath := TNoisyDeath.create;
  try
    waitAWhile;
  finally
    NoisyDeath.Free;
  end;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  NoisyDeath: TNoisyDeath;
  girm: TmtReaper;
begin
  NoisyDeath := TNoisyDeath.create;
  girm := TmtReaper.Create(NoisyDeath);
  waitAWhile;
end;

end.
```

我们通过分析 frmMain 单元, 可以看到 mtReaper 单元接口的使用方法。注意到 TNoisyDeath 对象覆盖了它的析构函数, 使其调用 ShowMessage 方法, 因此我们可以知道它被销毁的准确时间。WaitAWhile 方法负责计数直到 5000, 并在窗体的标题栏上显示计数值, 目的是让我们有足够长的时间看到对象被销毁时的一些情景。为了比较, 这里设计了两个按钮 Button1 和 Button2。在 Button1 的 Click 事件中, 我们按照常规方法创建并销毁 NoisyDeath 对象。而在 Button2 的 Click 事件中, 我们创建 NoisyDeath 对象后把该对象的引用传递给了 TmtReaper 的 FObject 变量, 对象传递是在 TmtReaper 的构造函数中实现的, 这意味着 NoisyDeath 对象和 TmtReaper 类型的 girm 对象有着相同的生命期。一旦 Button2Click 方法结束, TmtReaper 类型变量 girm 的作用域也将结束, 这导致引用计数器递减。此时引用计数器值减至 0, TmtReaper 对象被销毁, NoisyDeath 对象也就会随之销毁。Button2Click 方法中没有 try...finally 块, 没有内存泄漏, 代码输入量比较少, mtReaper 单元可以被反复使用以实现自动垃圾收集功能。

大家不要小看这个简单的例子, 里面包含着许多有用的思想。首先是设计模式的思想, 它用到了一种称为 Decorator 的设计模式。Decorator 模式可以动态地给一个对象添加一些额外的职

责。就扩展功能而言，Decorator 模式比生成子类方式更为灵活。为什么这么讲？试想，如果我们为了利用接口自动管理对象实例的好处，而让自己的类通过实现某个接口来获得这一功能，那么就要为所有定制的对象构建接口，同时也要创建子类以显露所有 VCL 对象和第三方控件中接口的能力。这会使得子类数目呈爆炸性增长，增加工作量。所以，我们为分离到其他对象中的这个对象委派职责，它们的生命期由接口来管理。为此，才创建了一个实现接口的普通对象变量，并将需要管理生命期的那个对象引用传递给它，也就是说让它管理。这种方法不但适用于任何对象，而且还减少了编程工作量。

记住，Decorator 模式经常会在以下情况中使用：

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；另一种情况可能是，因为类定义被隐藏，或类定义不能用于生成子类。

另外，这个例子给了我们这样一个启示：既然可以利用接口的实例计数器增减功能，让该接口实现自动清除垃圾的功能，那么也可以在接口实例计数器增减的时候干点别的什么事，比如保存和恢复的一些状态。

大家还记得我们前面在示例程序 3-5 和示例程序 3-6 中实现了一个保存和恢复字体类型 (TFont) 对象状态的例子吗？我们可以用接口的另类用法来进一步改写这个示例程序，使之成为一个模拟对象状态改变后自动恢复的例子，如示例程序 6-12 所示。

示例程序 6-12 一个模拟对象状态改变后自动恢复的例子

```
unit uSnapshot;  
  
interface  
  
uses SysUtils, mtReaper, classes;  
  
type  
  ISnapshot = interface  
    ['{FA256FA8-211F-462D-890B-FC0EB6096AD8}']  
    procedure Restore;  
  end;  
  
  TSnapshot = class (TInterfacedObject, ISnapshot)  
  private  
    FOriginal: TPersistent;  
    FTarget: TPersistent;  
    FReaper: ImtReaper;  
  public  
    constructor Create (Target: TPersistent);  
    destructor Destroy; override;  
    procedure Restore;  
  end;
```

```
implementation

constructor TSnapshot.create (Target: TPersistent);
begin
  FOriginal := TPersistent (Target.classType.create);
  FReaper := TmtReaper.create (FOriginal);
  FTarget := Target;
  FOriginal.Assign (Target);
end;

destructor TSnapshot.Destroy;
begin
  restore;
  Inherited;
end;

procedure TSnapshot.Restore;
begin
  if FTarget <> nil then
    FTarget.Assign (FOriginal);
end;
end;

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls;

type
  TForm1 = class (TForm)
    FontDialog1: TFontDialog;
    btnSet: TButton;
    Memo1: TMemo;
    procedure btnSetClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
    procedure WaitAWhile;
  private
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation

uses uSnapshot;

{$R *.dfm}
```

```

procedure TForm1.btnSetClick (Sender: TObject);
var
  FontSnapshot: ISnapshot;
begin
  FontSnapshot := TSnapshot.Create (Mem1.Font);
  {这里可以填写可能会改变 TFont 状态的任何代码}
  if FontDialog1.Execute then
    Mem1.Font := FontDialog1.Font;
  Mem1.Update;
  WaitAWhile;
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
  Mem1.Lines.Add (
    '一个模拟对象状态改变后自动恢复的例子。');
end;

procedure TForm1.WaitAWhile;
var i: Integer;
begin
  for i := 0 to 5000 do
  begin
    caption := '状态恢复倒计时: ' + IntToStr (5000-i);
  end;
end;

end.

```

这个程序的项目文件名称是 Snapshot.dpr，包含了三个程序单元，分别是：提供自动回收垃圾对象的 mtReaper、用于自动保存和恢复对象状态的 uSnapshot 和演示窗体 Unit1。示例程序 6-12 显示了 uSnapshot 和 Unit1 的全部源代码，mtReaper 的代码与示例程序 6-11 中的相同。这个程序的所有文件可以在随书光盘上找到。

下面我先介绍用于自动保存和恢复对象状态的 uSnapshot 单元。在这个单元中我们可以通过克隆当前对象来获得该对象的一个快照 (Snapshot)，并在作用域结束时利用对象快照恢复原对象的状态。该单元包含了一个名为 ISnapshot 的接口，该接口有一个 Restore 方法。通过 TSnapshot 类我们实现了 ISnapshot 接口。另外 TSnapshot 类还有一个构造函数、一个析构函数。

示例程序 6-12 的代码是这样运行的：首先创建一个与目标对象类型相同的内部实例（快照对象），它可以用来克隆目标对象。为了使快照对象更具有通用性，我们调用了 Target.classType.create，并将其向上转型为 TPersistent 类型，然后保存在 FOriginal 变量中。这样，只要目标对象是 TPersistent 的派生类类型，并具有自己的 Assign 方法（比如：TFont、TPen、TBrush 等），就可以实现多态，并未经任何改动地复用 uSnapshot 单元代码。请注意，快照对象是临时的。当目标对象超出其作用域后，其改变的状态需要自动复原，快照对象在完成目标对象复原任务后应该自动销毁，为此我们将快照对象的引用传递给 ImtReaper 类型的变量 FReaper，以实现快照对象生命期的自动管理，也就是示例程序 6-11 所演示的那样。

TSnapshot 类的构造函数完成了创建快照对象、建立快照对象生命期管理任务、克隆目标对

象等一系列工作。而 `TSnapshot` 类的析构函数则在目标对象作用域结束后通过反克隆，恢复目标对象的原状态。这里用到了 `Assign` 方法，不明白的读者可以回顾 3.1.5 节的内容。

在用于演示的 `Unit1` 单元，通过按钮的 `btnSetClick` 事件，我们可以改变文本框中的字体样式。实际上是演示了 `Font` 对象状态的改变。为了让读者看清发生了什么，我们有意在 `btnSetClick` 事件中加入 `WaitAWhile` 方法，进行延时。读者可以发现，无论在 `btnSetClick` 过程中填写任何可能会改变 `TFont` 状态的代码，我们在窗体上都可以直观地发现，最后 `Font` 对象总能恢复到原来的状态。因为变量 `FontSnapshot` 的作用域仅限于 `btnSetClick` 过程中，而且它是 `ISnapshot` 类型。我们巧妙地利用接口实例计数器的功能自动管理了对象状态的变化！

在这里，你可能会感叹“`FontSnapshot := TSnapshot.Create (Mem1.Font);`”这行代码的简洁和神奇。仅仅从这一行代码中，我们是无法看到对象的复制、状态的保存和恢复以及临时对象自动销毁等一系列动作的。这就是封装的魅力！显然，以面向过程的陈旧思维是很难看懂示例程序 6-12 中的代码的，因为这个小小的例子中包含了设计模式、接口、封装、继承、多态、对象引用和生命期等丰富的面向对象思维。我们不得不承认是面向对象思维拓展了我们的想像力。

第7章 研究封装

7.1 什么是封装

封装 (encapsulation) 是面向对象中的一个重要概念。简单地讲, 封装就是把东西包装起来, 隐匿了与用户无关的内部复杂性。这样, 无论使用的对象如何复杂, 在使用者看来都是一个易于操作的“黑匣子”。封装提供了一系列重要的好处:

- 所有的内部实现细节对外隐藏 (信息屏蔽), 这将减少当变化发生时副作用的传播。
- 数据和操作合并单个命名的对象或实体中, 这将便于组件化的复用。
- 简化了被封装对象或实体间的接口, 使系统耦合度降低。

《面向对象方法: 原理与实践》的作者 Ian Graham 一语道破了封装的实质: “数据和处理过程结合在一起并隐藏在接口后面”。

7.1.1 封装的概念

如果说数据的抽象化和继承关系使得概念和定义可以复用; 多态性使得实现和应用可以复用; 那么, 抽象化和封装则可以保持和促使系统的可维护性。

在面向对象编程中, 对象的使用者通过预先定义的接口关联到某一对象的服务和数据时, 无需知道这些服务是如何实现的, 即用户使用对象时无需知道对象内部的运行细节。这样, 以前所开发的系统中已使用的对象能够在新系统中重新采用, 减少了新系统中分析、设计和编程的工作量。

由此可见, 封装实际上是利用了一种抽象机制来管理事物本身的复杂型。从历史上看, 面向对象的抽象机制是从函数到模块、从抽象数据类型到对象的抽象, 一步步自然发展而来。早期的函数和过程也为程序员提供了实现信息隐藏的能力。一个程序员写的函数可以被其他程序员使用。其他程序员不需要知道实现的具体细节, 他们只需要知道接口。但是, 抽象的函数并不是一种有效的信息隐藏机制。它们只是部分地解决了多个程序员使用相同名称的问题, 但没有方法来限制对全局名称的访问和其可见性。今天, 面向对象编程已经扩展了封装信息和操作的概念, 改进了管理事物复杂性的抽象机制, 使得对象之间的访问更加安全和简便。

然而在实际的面向对象开发中, 封装有着更广义的概念, 是一个十分有趣的话题。

首先, 封装有逻辑上和物理上的不同语义。在逻辑上, 封装是对某一特定逻辑功能块的封装, 这个特定的逻辑功能块可以是一个类, 也可以是一个包 (类的集合)。它们都有自己明显的逻辑边界, 即具备功能上的原子性。物理上的封装是指具体实现代码的物理集合, 它以动态链接的物理形式存在。具体可以表现为 bpl 包 (bpl 包是 Delphi 特殊编译的 DLL)、动态链接库 (DLL) 或组件包 (COM+) 等形式。物理上的封装和逻辑上的封装没有一一对应的要求, 但它们之间存在必然的联系。

封装有粒度的概念, 即在不同的功能层次上都存在各自对应的封装要求。封装的粒度从

类、包（类的集合）、模块、子系统到系统依次增大。在逻辑上，这种层次化结构的粒度划分有利于管理系统的复杂性。但在物理上，封装粒度的划分多是从系统（地理）分布和易于维护的角度考虑，比如传统的桌面系统和瘦客户机的 C/S 系统就存在明显的不同，所以物理上的封装粒度划分无需和逻辑上的划分保持一致。

另外，无论是逻辑上的还是物理上的封装，其原则都是简化用户接口，隐藏实现细节。一个设计良好的接口，应该是稳定的、冗余最小的、功能完善的。使用接口的目的是为了将实现的细节分离出来，便于封装。这样，即使是封装的内容（即实现细节）发生了改变，稳定的接口也可以使得其他与之耦合的功能模块不会受到影响，从而保护其他代码不需随之改变。尽量让接口冗余最小并且不能以牺牲接口功能为代价。一方面，要使接口覆盖所有实现的功能，否则不能实现完善功能的封装将失去其封装的意义；另一方面要尽量使得接口简洁易用，避免接口冗余，否则会导致接口利用率不高，使用效率下降。所以封装的另一个关键在于接口的设计。好的封装必然有好的接口！

读到这里，读者可能已经明白我在本章要讨论的封装不仅限于类级别的封装上，我们还会研究封装在面向对象开发中更广泛、更实用的一些方面。

7.1.2 切割和封装的原则

有经验的程序员都知道，要创建一个可维护的应用程序，首先应该将应用程序切割成各个小块，然后分别开发并封装成独立的模块，最后组合成一个完整的系统。显然经过封装的模块可以隐匿复杂性、降低耦合度、提高复用率。Delphi 提供了多种切割程序的方式，你可以把程序代码分成类/类型（包括：form、datamodules、webmodules 等）、单元（.pas 文件）、项目/项目组，从而编译成组成应用程序的可执行文件、动态链接库或 COM + 组件等。如何切割和封装应用程序呢？我们通常需要从以下几个方面去考虑：

- 内聚力（cohesion）
- 耦合度（coupling）
- 间接性（indirection）

1. 内聚力

内聚是事物内部相互聚集相互依赖的一种紧密关系，内聚力反映了这种关系的状态和程度。被封装的程序模块内部内聚力越强越好。一个高内聚力的模块，其中的组成部分（类、方法、数据等）之间的关系是密不可分的。这种紧密的关系，体现了一种分割上的原子性。高内聚力的模块使得程序在修改时，只需改动该模块中的代码，而不会影响其他模块和整个系统。

比如在 Delphi 中我们经常用到的数据模块（DataModule）就是一个与内聚力有关的例子。有些程序员不太喜欢用数据模块，他们认为把 TTable 或 TQuery 组件直接放在 form 上而使用更方便，但这样的程序却不好维护。

首先，当有多个 form 需要存取相同的数据库时，就可能为每个 form 放上一些相同作用的 TTable 或 TQuery 组件，编写类似的事件处理程序。其次，form 所属的单元会夹杂着大量与界而无关的程序编码，难以理清和区分界面和业务逻辑，使得功能模块的封装变得困难。本来 form

只负责程序界面工作，数据模块负责数据库的存取和连接，但由于职责不清和代码混乱，导致了它们自身内聚力的下降，不利于封装和维护。

还有一种违背内聚力的设计，就是将整个系统的所有数据库存取组件（TTable、TQuery 等）都放在一个数据模块中，即整个应用程序只有一个数据模块（我见过最糟糕的一个数据模块中有各种数据库组件 100 多个）。对于一个复杂的数据库应用程序而言，显然这不是一个好的设计。因为仔细分析后可以发现其中还有进一步分割以提高内聚力的可能。如果将一个大数据模块分割成负责处理发票、订单、客户资料等若干个小数据模块则更便于使用和维护。

参见 第 4 章中讨论了一个动态创建数据模块的数据库应用程序。图 4-17 提供一种数据模块的设计方案，图 4-18 是这个设计的 UML 类图。

即使涉及的数据库不是很多，若将一个数据模块分成带事务（Transaction）的（负责更新数据库）和不带事务的（负责查询和浏览数据库），亦可以提高设计上的内聚力。比如前者可以封装成需要事务的 COM+ 对象，后者封装成不需要事务的 COM+ 对象，有利于提高系统的运行效率。

2. 耦合度

耦合是指两个（或两个以上）的系统之间通过相互作用而彼此影响的现象。在程序设计中，模块之间的耦合度应该越弱越好，这样模块之间互相牵制和依赖的作用就变小，便于模块的维护和重用。通常降低耦合度的方法是通过一种规范的通信机制来保持模块之间的联系（比如对象间的消息传递）。相反，滥用全局变量的面向过程编程习惯则会导致模块之间耦合度增加。比如，下而这段代码：

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form2.Edit1.Text := 'test';  
end;
```

上述代码直接使用其他 form 上的组件（该单元 uses 了 Form2 的单元），虽然可以运行，但却不易维护。如果一旦改动了 Form2 的 Edit1，将其变为 LabeledEdit1，就需要在 Form1 单元中找到所有涉及 Form2.Edit1 的代码进行改动。如果 Form2.Edit1 仅仅出现在 Form1 单元中，影响还不小，但是类似的代码如果出现在多个地方，比如远程服务器程序中，麻烦就大了。这种通过单元之间互相 uses 的全局变量的使用所带来的后果会使你无法确定哪些地方在维护时没有改到。显然，这种紧耦合使得模块之间难以维护。解决这个问题的方法是通过接口实现模块之间数据的间接存取，即将强耦合的直接作用化做弱耦合的间接作用。

3. 间接性

间接性是指模块之间避免数据的直接存取，而要通过接口进行中介和缓冲。如果模块之间的程序变量可以互相调用，数据可以直接相互改写，那么模块就失去了独立性，无法进行封装。间接性的特征主要体现在中介和缓冲上。所谓中介，就是通过接口来提供外界访问的途径，而将内部的程序屏蔽起来，使得内外有别。这里接口是泛指。对于 Delphi 程序单元而言，它可以分成接口部分（interface）和实现部分（implementation），只有接口部分的变量或类才能被其他单元访问。对于类而言，公有的（public）或公布的（published）类成员（比如属性）是

接口，可被外界访问。对面向对象的设计而言，我们将对象接口或 COM 接口作为模块的接口，提供给其他模块使用。设计良好的接口只提供中介的作用，并不提供复杂的功能实现，所以当实现代码发生了变化时，它能够通过自身的缓冲来匹配、调整这种变化。

前面我们提到的那个例子可以改成这样：

```
type
  TForm1 = class (TForm)
    Button1: TButton;
    procedure Button1Click (Sender: TObject);
    .....
  public
    procedure SetXXXText (Value: string);
  end;
.....

implementation

uses Unit2;

{$R *.dfm}

procedure TForm1.SetXXXText (Value: string);
begin
  Form2.Edit1.Text := Value;
end;

procedure TForm1.Button1Click (Sender: TObject);
begin
  SetText ('Test');
end;

.....
```

通过 SetXXXText 方法提供的中介，使得程序中原来对 Form2.Edit1.Text 直接访问的地方，统统改变成了对 SetXXXText 方法的调用。今后一旦 Form2 的 Edit1 变为 LabeledEdit1 或别的什么，我们只需在 SetXXXText 方法这一个地方进行修改就可以了。

```
procedure TForm1.SetXXXText (Value: string);
begin
  Form2.LabeledEdit1.Text := Value;
end;
```

这是一个简单的例子，但说明的道理是一样的。在复杂的应用中，可能这个中介不一定是一个方法，而是一个抽象类或 COM + 接口，甚至是一个中间层协调对象。无论如何，它们都是体现了间接性原则。

7.2 逻辑上的封装

7.2.1 类的封装

所谓封装就是指封装系统的可变因素，并以一个相对稳定的接口来提供服务。在面向对象

的语言中,类是一种最好的逻辑封装形式。通过类的封装,可以把可变性封装到一个类中,避免了把可变性散落在代码的很多角落里。对于同一种可变性的不同表象,可以通过继承结构中的具体子类来解决。但在使用类来封装可变性时,不能将一种可变性和另一种可变性混合在一起,这意味着类的继承结构不易过深,一般宜在两到三层。

类级别的封装是最常见的逻辑上的封装形式。对于 Delphi 的类,有不同的访问级别,分别为 published (公布的)、public (公有的)、protected (保护的)、private (私有的)。published 和 public 区别不大,其成员都可以被外界代码直接访问。因此,在类级别的封装中,对外的接口是 public 的方法和 published 的事件和属性。而 protected 和 private 成员则属于类的实现细节,前者可以被该类的所有派生类访问,后者仅有类本身和友元(同一单元中的其他类)可以访问。

通过合理限制类成员的可见性,可以完成类的封装,实现对类的安全访问。由于类本身的设计就已经将对象的数据和对数据的操作封装在一起,所以,我们千万不能将类的数据成员视为普通的变量,进行直接的修改。Delphi 提供了属性来封装数据,虽然形式上我们看不到 setXXX 和 getXXX 来读写数据,但本质上却是一样通过方法来修改数据。记住,所有放在私有段中的成员数据都必须通过 Property 来访问是体现封装的好风格。

下面一段代码是我们最常见的代码,这段代码虽然没有语法错误,但严格按照面向对象的思维来考虑,还是存在问题:

```
unit Unit1;

uses
  Windows, ..., Unit2;
  //这里 Use 了 Unit2,使得 Unit2 中的 Form2 作为一个全局变量使用。

type
  TForm1 = class (TForm)
    Button1: TButton;
    ... ..
  var
    Form1: TForm1;

  implementation

  procedure TForm1.FormCreate (Sender: TObject);
  begin
    Form2 := TForm2.Create (self);
  end;

  procedure TForm1.Button1Click (Sender: TObject);
  begin
    Form2.Show; // < - 这里把 Form2 作为一个全局变量使用。
  end;

  procedure TForm1.Button2Click (Sender: TObject);
  begin
    Form1.caption := 'Hi'; // < - 不要在 TForm1 类中使用 Form1。
  end;
```

```
//-----
unit Unit2;
....
var
  Form2: TForm2; // < - 这里把 Form2 声明为一个全局变量。
....
implementation
....
```

首先，这个例子中程序员将 Form2 作为一个全局变量使用，破坏了 TForm1 这个类的封装性。

其次，Form2 是 TForm2 的一个实例引用，它居然被写死到 TForm1 的类中，成为 TForm1 类 Button1Click 方法实现的一部分。由于这个 Form2 的存在，使得 TForm1 类失去了其封装性。对于大多数情况而言，在一个项目中，TForm1 和 TForm2 只可能各有一个实例，所以这样的代码也能勉强通过。但是从严格意义上来说，也是不符合封装性的要求。

另外，应该避免在类的方法中使用一个特定的对象名称，换句话说，不应该在 TForm1 类的方法中直接使用 Form1。如果确实需要使用当前的对象，可以使用 self 关键字。请牢记：大多数时候都没有必要直接使用当前对象的方法和数据。如果不遵循这条规则，当你为一个窗体类创建多个实例的时候，就会陷入麻烦当中。Marco Cantu 有句名言：“Never Use Form1 in TForm1”。

这个比较具有普遍性的例子暴露了一些编程人员概念上的混乱，其核心就是封装的问题。我们将代码重新修改如下：

```
type
  TForm1 = class (TForm)
    Button1: TButton;
    procedure Button1Click (Sender: TObject);
  private
    {Private declarations }
    FForm: TForm; // FForm 是 TForm1 的私有成员。
  public
    {Public declarations }
    property FForm: TForm read FForm write FForm; //通过属性访问 FForm。
  end;
var
  Form1: TForm1;

implementation
uses Unit2;
{$R *.dfm}

procedure TForm1.Button1Click (Sender: TObject);
begin
  if Assigned (FForm) then
    TForm2 (FForm) .Show; //访问的是内部成员 FForm,注意 FForm 需要转型。
end;

procedure TForm1.Button2Click (Sender: TObject);
begin
```

```
    self.caption: = 'Hi'; // < - 在 TForm1 类中使用 self 代替 Form1。
end;

end.

// 以下是项目文件中的内容。
program Project1;
uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1},
    Unit2 in 'Unit2.pas' {Form2};
{$R *.res}
begin
    Application.Initialize;
    Application.CreateForm (TForm1, Form1);
    Application.CreateForm (TForm2, Form2);
    //通过属性传递 Form2 的引用。
    Form1.FForm := Form2;
    Application.Run;
end.
```

这段修改过的程序已经符合面向对象的封装要求。我们在 program Project1 中，分别创建了 TForm1 和 TForm2 的实例，并通过属性传递 Form2 的引用给 Form1 的内部成员 FForm，实现 TForm1 和 TForm2 之间的耦合。这样，当 Form1 在自己的方法中（比如 Button1Click 方法）通过 FForm 调用 Form2 的引用时，就遵守了封装性的原则！

当然，这些代码仅仅是为了体现封装的思想，而在实际开发中封装的程度可以灵活掌握，只要你有这种面向对象的思想就行。

最后我再强调以下两点，以加深大家对变量和属性的体会：

1) 尽量不要用全局变量。

即使要用，也要用全局对象来代替它。习惯了面向过程编程的程序员在这点上很容易犯错，而且这一点相对来说也比较难掌握。实际上，全局变量也是面向过程编程技术的一个很大的缺陷，难跟踪，难调试，也就难维护。为什么要用对象而不是变量呢？对象可以封装对变量的操作，任何对该变量的操作都必须通过调用对象的方法来完成，我们可以在操作该变量的方法中设置断点来调试，这就解决了前面所提到的 3 个难点（难跟踪、难调试、难维护）。

如果需要为窗体存储额外的数据，则可以向窗体类中添加一些私有数据成员。在这种情况下，每一个窗体实例都会有自己的数据副本。你可以使用在单元的 implementation 部分定义的变量（即单元变量），声明那些供窗体类的多个实例共享的数据。如果你需要在不同类型的窗体之间共享数据，就可以把它们定义在主窗体里来实现共享，或者使用一个全局变量，使用方法或者是属性来获得数据。

2) 对象之间交换数据，尽可能地使用属性而不是变量。

为什么要用属性而不是变量呢？对数据的操作可以通过属性的读写方法进行封装，一旦以后对象内部的数据结构发生了变化，只要我们提供的属性接口不变，对程序别的部分的影响就能减小到最小。例如 Form 之间通过属性来交换数据。以后因为某个原因你要将原来用数组实现的东西改为用链表实现，则只要属性接口仍然是数组，那对别的对象就几乎没有影响。

7.2.2 数据的封装

1. 用属性封装数据

在面向对象编程的思维中，我们用方法描述对象的行为，用数据描述对象的状态，而属性则通过 property 读写限定符对私有数据提供了受限制的访问途径，保护了数据免受破坏。属性可以说是 Delphi 的一大特色，在 C++ 中程序员还得使用 GetXXX 和 SetXXX 来访问私有数据成员，但是 Delphi 的属性用起来感觉爽多了，因为连 GetXXX 和 SetXXX 都被封装了。这一点很快被微软“学”去了，C# 中就引入了属性的概念。

把数据成员放在私有段中，并通过 property 来访问是个好的编程风格，这样只要类内部的数据使用正确，那么数据是不会被破坏的。

例如下面的写法中，将存储年龄的数据成员放在公有段中直接供外界访问就很不安全，因为这里的 FAge 就好像是一个全局变量可以任意修改，难免会出现 1000 岁的笑话。

```
type
  TMan = class (TObject)
  public
    FAge: Integer;
    .....
```

如果改为属性方式访问，情况就大不一样。

```
type
  TMan = class (TObject)
  private
    FAge: Integer;
    function GetAge: Integer;
    procedure SetAge (Value: Integer);
  public
    property Age: Integer read GetAge write SetAge;
    .....
```

这样一来，我们就可以在属性 Age 的读写方法中添加安全控制代码，限制非法操作。而编程过程中使用属性 Age 作为操作数，感觉上和使用一个普通变量一样方便。

由于属性 Age 封装了年龄的读写规则，因此，规则变化时我们可以在类中随时修改和升级规则代码，而不必改动其他程序。下面就是一个写年龄的规则，将年龄大小限制在 0~120 岁以内，否则进行错误标记。

```
procedure TMan.SetAge (Value: Integer);
begin
  if (Value > 0) and (Value < 120) then FAge := Value
  else FValidData := False;
end;
```

在有些情况下，利用写规则还可以限制一些不必要的数据库更新，以免浪费 CPU 时间、网络带宽等宝贵资源。比如写入到图形控件的 Picture 特性的图形、写入到数据库字段的大数据都可以利用写方法进行检查，以确保仅对新的或改变的数据进行更新。例如：

```
procedure TMan.SetPhoto (Value: TPicture);
```



```

begin
  if (FPhoto.Picture = Value) then Exit;
  FPhoto.Picture.Assign (Value);
  Repaint;
end;

```

示例程序 7-1 显示了用属性封装数据的 TMan 类。

示例程序 7-1 用属性封装数据的 TMan 类

```

unit uCreateManClass;

interface

uses
  Dialogs;

type
  TSkinColor = (scWhite, scYellow, scBlack, scDark); // '白色', '黄色', '黑色', '深色'
  TMan = class(TObject)
  private
    FAge: Integer;
    FLanguage: string;
    FMale: Boolean;
    FName: string;
    FSkinColor: TSkinColor;
    FValidData: Boolean;
    function GetAge: Integer;
    function GetColor: TSkinColor;
    function GetLanguage: string;
    function GetMale: Boolean;
    function GetName: string;
    procedure SetAge (Value: Integer);
    procedure SetColor (Value: TSkinColor);
    procedure SetLanguage (Value: string);
    procedure SetMale (Value: Boolean);
    procedure SetName (Value: string);
  public
    function retrieve: Boolean;
    function save: Boolean;
    procedure sayHello (words: pchar);
    property Age: Integer read GetAge write SetAge;
    property Language: string read GetLanguage write SetLanguage;
    property Male: Boolean read GetMale write SetMale;
    property Name: string read GetName write SetName;
    property SkinColor: TSkinColor read GetColor write SetColor;
    property ValidData: Boolean read FValidData write FValidData;
  end;

implementation

|
| ***** TMan
|

```



```
function TMan.GetAge: Integer;
begin
    result: = FAge;
end;

function TMan.GetColor: TSkinColor;
begin
    result: = FSkinColor;
end;

function TMan.GetLanguage: string;
begin
    result: = FLanguage;
end;

function TMan.GetMale: Boolean;
begin
    result: = FMale;
end;

function TMan.GetName: string;
begin
    result: = FName;
end;

function TMan.retrieve: Boolean;
begin
end;

function TMan.save: Boolean;
begin
end;

procedure TMan.sayHello (words: pchar);
begin
    showmessage (words);
end;

procedure TMan.SetAge (Value: Integer);
begin
    if (Value > 0) and (Value < 120) then FAge: = Value
    else FValidData: = False;
end;

procedure TMan.SetColor (Value: TSkinColor);
begin
    if (Value < > FSkinColor) then FSkinColor: = Value;
end;

procedure TMan.SetLanguage (Value: string);
begin
    if (Value < > FLanguage) then FLanguage: = Value;
end;
```

```

procedure TMan.SetMale (Value: Boolean);
begin
  if (Value < > FMale) then FMale: = Value;
end;

procedure TMan.SetName (Value: string);
begin
  if (Value < > FName) then FName: = Value;
end;

end.

```

其实，可以用属性来封装的数据类型很多，甚至可以通过属性来访问一个对象。这一点是非常有意义的。在示例程序 7-2 中，我们可以通过属性 `Data _ Man` 来访问一个 `TMan` 类型的对象，并通过读方法 `GetData _ Man` 和写方法 `SetData _ Man` 将这个对象的属性从一个数据集中读出和写入。

示例程序 7-2 通过属性来访问对象

```

unit DataProcess;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, DBClient, uCreateManClass;

type
  TCDSMan = class (TClientDataSet)
  private
    FMan: TMan;
    function GetData _ Man: TMan;
    procedure SetData _ Man (Value: TMan);
  public
    constructor create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Data _ Man: TMan read GetData _ Man write SetData _ Man;
  end;

implementation

{
  * * * * * TCDSMan
}

constructor TCDSMan.create (AOwner: TComponent);
begin
  inherited;
  LoadFromFile ('ManData');
  open;
end;

```

```

destructor TCDSMan.Destroy;
begin
    FMan.Free;
    SaveToFile (' ManData ');
    inherited;
end;

function TCDSMan.GetData _ Man: TMan;
begin
    if FMan = nil then FMan := TMan.Create;
    //set FMan
    if RecordCount > 0 then
    begin
        FMan.Name := FieldByName (' Name' ).Value;
        FMan.Age := FieldByName (' Age' ).Value;
        FMan.Language := FieldByName (' Language' ).Value;
        FMan.SkinColor := FieldByName (' SkinColor' ).Value;
        FMan.Male := FieldByName (' Male' ).Value;
    end
    else
    begin
        FMan.Name := '新用户';
        FMan.Language := '中文';
        FMan.SkinColor := scYellow;
        FMan.Age := 20;
        FMan.Male := True;
    end;
    result := FMan;
end;

procedure TCDSMan.SetData _ Man (Value: TMan);
begin
    if Value = nil then
    begin
        FMan.Free;
        FMan := nil;
        Exit;
    end;
    if not active then open;
    Insert;
    FieldByName (' Name' ).Value := FMan.Name;
    FieldByName (' Age' ).Value := FMan.Age;
    FieldByName (' Language' ).Value := FMan.Language;
    FieldByName (' SkinColor' ).Value := FMan.SkinColor;
    FieldByName (' Male' ).Value := FMan.Male;
    Post;
end;

end.

```

为了演示属性的神奇功能，我设计了一个程序，以利用属性实现数据集和对象之间的读写。该程序的设计包括外观类 TForm1、业务类 TMan 和数据访问类 TCDSMan，如图 7-1 所示。该程序包含 3 个单元，分别如示例程序 7-1、示例程序 7-2 和示例程序 7-3 所示。

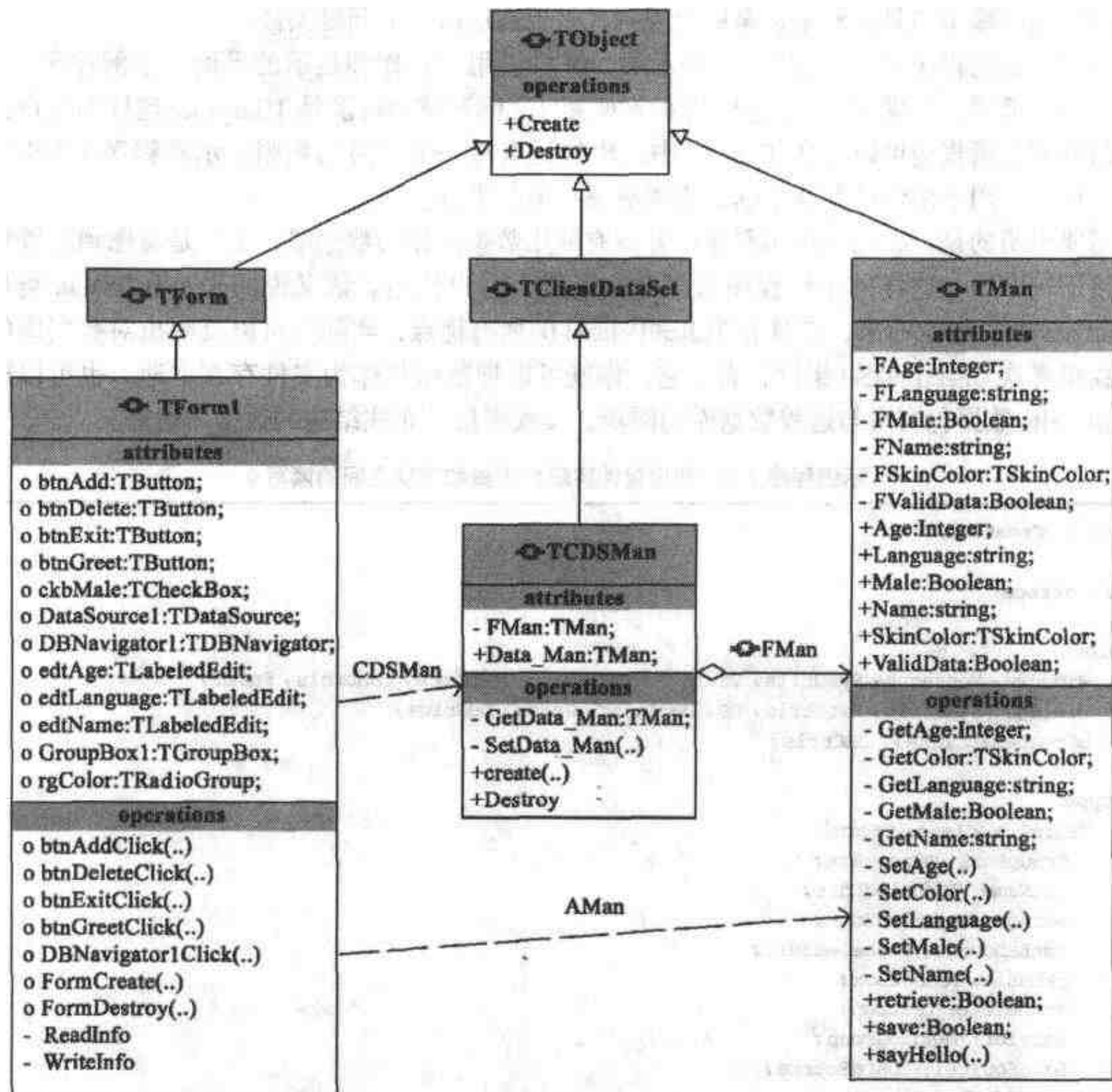


图 7-1 属性封装数据演示程序类图

由于示例程序 7-2 中 TCDSMan 类提供了读写 Data_Man 属性的方法，并在读写方法规则中实现了对数据集的访问，所以在示例程序 7-3 中只需简单的赋值语句就完成了数据集和对象之间的读写过程：

```

AMan: = cdsMan.Data_Man; //读数据
cdsMan.Data_Man: = AMan; //写数据

```

别小看这点，它可是迈向面向对象数据库开发的第一步。这种完美的封装除了代码简洁优美之外，还为我们提供了分层实现的概念。

一个设计良好的软件系统的体系结构通常分为三层：表示层、逻辑层和数据层。在表示层，常是一些具体与用户交互的对象，如：按钮、菜单和对话框。在数据层，则是从问题域中找出描述实体的表。完全面向对象和非完全面向对象的重大区别在于逻辑层。什么是逻辑层？

逻辑层是问题域中的具体对象，是商业规则，是更高层次上的问题实体。

这里，示例程序 7-3 相当于一个表示层，提供了用户操作和显示的界面。示例程序 7-1 相当于一个逻辑层，提供了 TMan 这样的业务对象（实际开发中可能是 TCustomer 这样的客户类），相关的所有逻辑规则可以写在 TMan 类中，比如这里有一个年龄的规则。示例程序 7-2 相当于一个数据层，用于维护持久性数据，完成数据存取的事务。

需要说明的是，在这个演示程序中并没有使用数据库作为数据层，主要是考虑到配置数据库连接比较麻烦，这样的示例程序太复杂，不便于用户使用。话又说回来，TCDSMan 类是从 TClientDataSet 类继承而来，它具有 TClientDataSet 的所有优点，实际上可以反映出对数据库的操作。我很喜欢 ClientDataSet 组件，有了它，你既可以把数据集作为文件存在本地，也可以通过 Data 和 Delta 数据包实现与远程数据库的同步，实现多层分布式结构的数据库运算。

示例程序 7-3 利用属性实现数据集和对象之间的读写

```
unit uCreateMan;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls, DB, DBClient, Grids, DBGrids,
  uCreateManClass, DBCtrls;

type
  TForm1 = class (TForm)
    GroupBox1: TGroupBox;
    edtName: TLabelledEdit;
    edtAge: TLabelledEdit;
    edtLanguage: TLabelledEdit;
    ckbMale: TCheckBox;
    btnDelete: TButton;
    rgColor: TRadioGroup;
    DataSource1: TDataSource;
    btnAdd: TButton;
    btnExit: TButton;
    DBNavigator1: TDBNavigator;
    btnGreet: TButton;
    procedure btnDeleteClick (Sender: TObject);
    procedure btnGreetClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
    procedure btnAddClick (Sender: TObject);
    procedure btnExitClick (Sender: TObject);
    procedure DBNavigator1Click (Sender: TObject; Button: TNavigateBtn);
    procedure FormDestroy (Sender: TObject);
  private
    procedure WriteInfo;
    procedure ReadInfo;
  public
    {Public declarations}
  end;
```

```
var
    Form1: TForm1;

implementation

uses DataProcess;

var
    AMan: TMan;
    CDSMan: TCDSMan;
{$R *.dfm}

procedure TForm1.ReadInfo;
begin
    AMan := CDSMan.Data_Man;
    edtName.Text := AMan.Name;
    edtAge.Text := IntToStr (AMan.Age);
    edtLanguage.Text := AMan.Language;
    rgColor.itemindex := ord (AMan.SkinColor);
    ckbMale.Checked := AMan.Male;
end;

procedure TForm1.WriteInfo;
begin
    AMan.ValidData := True;
    AMan.Name := edtName.Text;
    AMan.Age := StrToInt (edtAge.Text);
    AMan.Language := edtLanguage.Text;
    AMan.SkinColor := TSkinColor (rgColor.itemindex);
    AMan.Male := ckbMale.Checked;
    if AMan.ValidData then CDSMan.Data_Man := AMan
    else application.MessageBox ('输入了非法数据,请检查!', '提示信息', 0);
end;

procedure TForm1.btnDeleteClick (Sender: TObject);
begin
    DataSource1.DataSet.Delete;
    ReadInfo;
end;

procedure TForm1.btnAddClick (Sender: TObject);
begin
    WriteInfo;
end;

procedure TForm1.btnGreetClick (Sender: TObject);
begin
    AMan.sayHello ('你好!');
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
    CDSMan := TCDSMan.create (nil);
    DataSource1.DataSet := cdsMan;
```

```

    ReadInfo;
end;

procedure TForm1.btnExitClick (Sender: TObject);
begin
    close;
end;

procedure TForm1.DBNavigator1Click (Sender: TObject; Button: TNavigateBtn);
begin
    ReadInfo;
end;

procedure TForm1.FormDestroy (Sender: TObject);
begin
    CDSMan.Free;
    CDSMan := nil;
end;

end.

```

利用属性实现数据集和对象之间的读写，除了需要在数据集的字段值与对象的属性值之间建立对应关系，防止出现值类型不匹配或一方有值而另一方没有值的错误外，更要注意确保对象的存在和正确引用对象，否则会导致致命的错误。

为此，在 TCDSMan 的 GetData_Man 方法中必须严格检查 FMan 对象是否存在，以及数据集中是否有记录，并提供对应的措施。

```

function TCDSMan.GetData_Man: TMan;
begin
    if FMan = nil then FMan := TMan.Create; //FMan 对象不存在时
    //设置 FMan 属性
    if RecordCount > 0 then
    begin
        FMan.Name := FieldByName('Name').Value;
        FMan.Age := FieldByName('Age').Value;
        FMan.Language := FieldByName('Language').Value;
        FMan.SkinColor := FieldByName('SkinColor').Value;
        FMan.Male := FieldByName('Male').Value;
    end
    else
    begin //数据集中无记录时
        FMan.Name := '新用户';
        FMan.Language := '中文';
        FMan.SkinColor := scYellow;
        FMan.Age := 20;
        FMan.Male := True;
    end;
    result := FMan;
end;

```

同样，在 TCDSMan 的 SetData_Man 方法中必须严格检查作为参数传递的 TMan 对象引用是否存在，如果对象引用不存在，就不能向数据集的记录赋值。

```
if Value = nil then  
begin  
    FMan.Free;  
    FMan := nil;  
    Exit;  
end;
```

虽然示例程序 7-3 中使用了 TMan 对象变量 AMan, 示例程序 7-2 也使用了 TMan 对象变量 FMan, 但 TMan 对象只能在一个地方创建, 而且要保证这些对象变量引用的是同一个对象, 否则也会出错。为了便于 TMan 对象生命期的管理, 我们只在 TCDSMan 这一个地方创建和销毁该对象: 即在 TCDSMan 的 GetData_Man 方法中创建对象, 在 TCDSMan 的 Destroy 方法中销毁对象。一旦 TMan 对象创建, 就可以将数据集的记录赋值给对象的属性。有意思的是, 示例程序 7-3 的 TForm1.FormCreate 方法中, 调用 ReadInfo 方法不仅是为了读一条记录来初始化界面, 更重要的是通过 AMan := CDSMan.Data_Man 这条语句把 TMan 对象的引用通过 Data_Man 属性传递给 AMan 对象变量, 否则无法建立对象与界面之间的联系。

另外, 在示例程序 7-3 的 TForm1.FormCreate 方法中, TCDSMan 对象是手工创建的, 而不是一个拖放在 TForm1 上的组件 (这样可以可视化自动创建)。这里我没有将代码写成:

```
CDSMan := TCDSMan.create (self);
```

而是写成了:

```
CDSMan := TCDSMan.create (nil);
```

参见 如果还不清楚为什么使用 nil 代替属主对象, 则请参见 3.2.4 节的相关内容。

需要说明的是界面上没有使用数据感知 (data aware) 控件来存取数据。显然这是一个面向对象的数据库示例程序, 虽然简单, 但却不同于拖放数据库控件的那种 RAD 设计。

现在运行程序, 如图 7-2 所示, 我们不但实现了数据集和对象之间的读写, 还看到一个三层的面向对象的数据库应用系统的雏形。

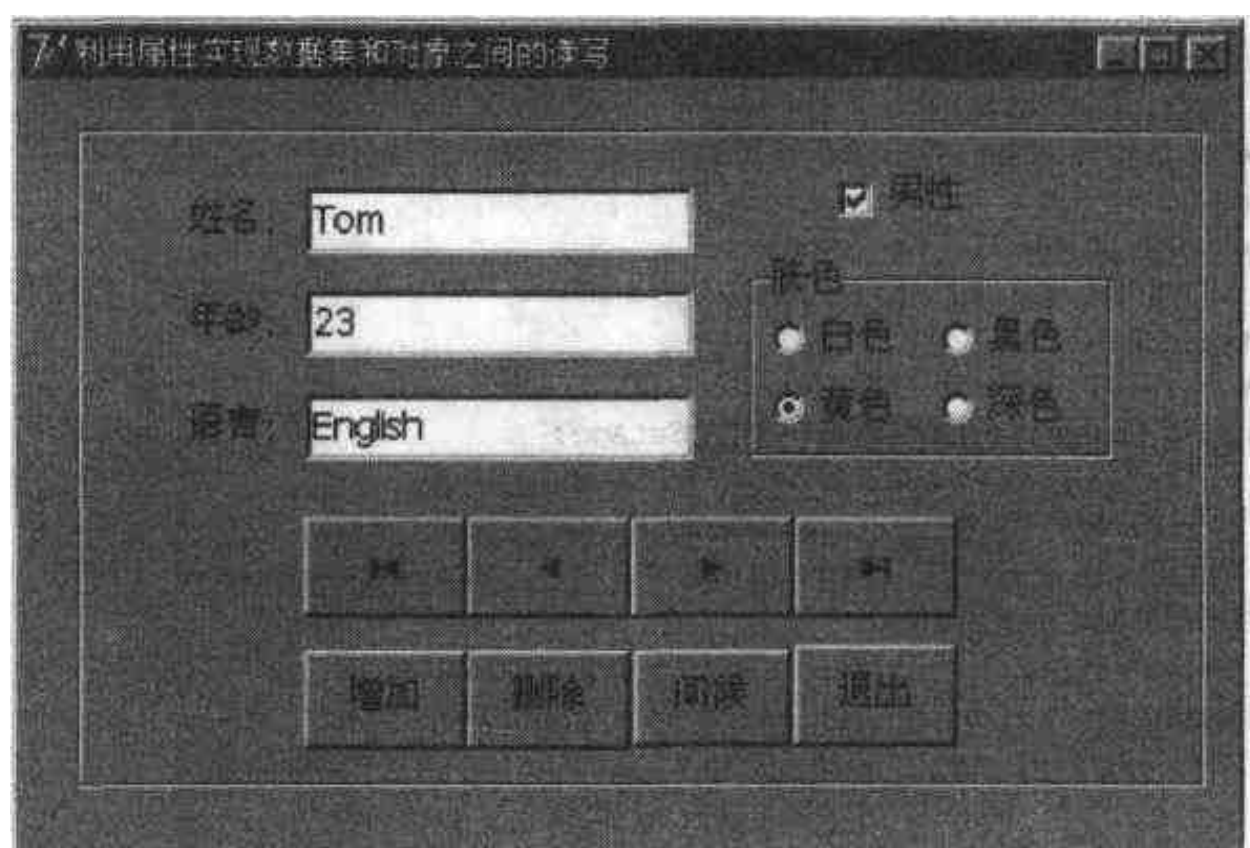


图 7-2 运行程序, 实现数据集和对象之间的读写

2. 使用索引属性

索引属性使用了索引限定符，它的特点是可以使几个属性共用同一个方法来表示不同的值。索引限定符包含 index 指示字，并在后面跟一个介于 -2147483647 到 2147483647 之间的整数常量。索引属性的读写限定符必须是方法而不能是数据成员。

索引属性并不是类似数组的单个属性，而是与多个具有相同的读写方法的属性相似。例如前面我们提到的 TMan 有一些元素都是字符型的，诸如：姓名、语言等，这些相同类型的元素读写规则也相同时不妨考虑将它们合并，以提高编程效率。最好的办法是使用索引属性来访问每个字符串的实际数值。

举例说明，我们可以将示例程序 7-1 改写为示例程序 7-4：

示例程序 7-4

```

unit uCreateManClass;

interface

uses
  Dialogs, Classes;

type
  TSkinColor = (scWhite, scYellow, scBlack, scDark); // '白色', '黄色', '黑色', '深色'
  TMan = class(TObject)
  private
    FAge: Integer;
    FData: TStrings; // FData 属性是一个 TStrings 对象, 用 "名称 = 值" 对形式存储数据
    function GetData (Index: Integer): string; // 索引属性统一的读方法
    procedure SetData (Index: Integer; Value: string); // 索引属性统一的写方法
    FMale: Boolean;
    FSkinColor: TSkinColor;
    FValidData: Boolean;
    function GetAge: Integer;
    function GetColor: TSkinColor;
    function GetMale: Boolean;
    procedure SetAge (Value: Integer);
    procedure SetColor (Value: TSkinColor);
    procedure SetMale (Value: Boolean);
  public
    constructor create;
    procedure sayHello (words: pchar);
    property Age: Integer read GetAge write SetAge;
    property Male: Boolean read GetMale write SetMale;
    property SkinColor: TSkinColor read GetColor write SetColor;
    property ValidData: Boolean read FValidData write FValidData;
    // 索引属性, 不同的属性用索引值区分
    property Name: string index 0 read GetData write SetData;
    property Language: string index 1 read GetData write SetData;
    property Nationality: string index 2 read GetData write SetData;
  end;

```

```

implementation

const
  DataName: array [0..2] of string = ('Name','Language','Nationality');

constructor TMan.create;
begin
  FData := TStringList.Create;
end;

//索引属性实现有着共同的读写方法
function TMan.GetData (Index: Integer): string;
begin
  result := FData.Values [DataName [Index]];
end;

procedure TMan.SetData (Index: Integer; Value: string);
begin
  FData.Values [DataName [Index]] := value;
end;

.....

end.

```

在示例程序 7-4 中，我们使用了索引属性来管理 Name、Language 以及 Nationality 属性，每个属性的使用都解析到 GetData 和 SetData 方法的调用。私有数据成员 FData 是一个 TStringList 对象，用“名称 = 值”对形式存储数据，这样属性和值就可以建立一一对应的关系，共同存储在 FData 中。因为对于 TStringList 中的所有对象而言，读写数据的代码都是相同的，所以无论是对于 3 个属性或是更多的属性，GetData 和 SetData 方法就足够了。显然，索引属性的使用，帮助我们省去了为每个属性分别撰写的重复代码。

注意 TStringList 是一个抽象类，抽象类在类中定义有抽象方法，这些抽象方法要由派生类声明相应的覆盖函数来实现，因此不能创建抽象类的实例。TStringList 类是 TStringList 类的派生类，所以我们在 TMan 的构造函数中要创建一个 TStringList 实例：

```
FData := TStringList.Create;
```

在 SetData 方法中，FData.Values [DataName [Index]] := value 将通过索引查找对应属性的值。比如：DataName [Index] 的值为 0 时，相当于 FData.Values ['Name'] := value，于是在 FData 中能够找到“Name = ”开头的字符串值，将其改写为 value 参数所传递的新值。这里的 Index 是索引参数，当它为 0 时显然这个 SetData 方法就成了 Name 属性的写方法了。因为 Name 属性是这样声明的：

```
property Name: string index 0 read GetData write SetData;
```

我们把 Name、Language 等属性改为索引属性，实际上并不需要改动示例程序 7-2 和示例程序 7-3 的代码。因为属性名称还是原来的 Name 和 Language，只有新增的 Nationality 属性需要加进对应的代码。

7.3 物理上的封装

7.3.1 物理封装和动态链接

物理上的封装意味着将程序封装成若干个独立的物理组成部分，各部分之间通过动态链接共同完成系统的功能，而且各个物理组成部分可以单独维护和编译，不影响其他部分。要理解物理封装首先要搞清楚静态链接和动态链接。

在 Delphi 中，如果程序的各个模块分别保存在不同的单元文件中，并通过 `uses` 指令来互相调用，这就是一个典型的静态链接。于是各个静态的子例程编译之后，连接器从 Delphi 编译过的单元（或静态库）中取出子例程编译代码并添加到执行文件中。最终 EXE 文件包括了程序及其所属单元的所有代码。显然，静态链接的单元或模块最终以一个独立的物理形式（可执行文件）存在。除了自己编写的单元文件外，Delphi 还自动 `uses` 了一些预设的单元，如：Windows、Messages 等，这些都是静态链接。

静态链接无法实现物理上的切割和封装，而且一旦其中某个单元或模块改动，其他所有单元或模块都得随之重新编译和连接。

用于实现物理切割和封装的 bpl 包、DLL 动态链接库或 COM+ 组件都是一种动态链接的形式。在动态链接情况中，连接器只使用子例程 `external` 声明中的信息在执行文件中产生一些数据表格。当 Windows 向内存中装载执行文件时，它首先装载所有必需的 DLL，然后程序才会启动。在装载过程中，Windows 用函数在内存中的地址填充程序的内部表格。

每当程序调用一个外部函数时，它就会使用该内部数据表格直接对 DLL 代码（它当前装载在程序的地址空间中）进行调用。注意，该模式不会涉及两个不同的应用程序，DLL 已经变成了应用程序的一部分，并装载在同一地址空间。所有参数的传递都发生在堆栈上，与其他任何函数调用一样。这里我们并没有讨论 DLL 的编译，因为我们首先想重点介绍两种不同的连接机制。

早期程序员较多地使用了 DLL，但是 DLL 在使用中还有不尽如人意之处。比如，当我们在主程序中调用 DLL 时，必须明确指定 DLL 的位置，否则会在执行中出错；当主程序调用不同语言开发的 DLL 时会存在问题，必须经过复杂的转换过程；主程序对 DLL 版本的依赖会导致很多问题。所以 COM/COM+ 的出现可以解决不少问题。

COM/COM+ 提供了包含方法和属性的接口，这些接口是负责控制整个 COM/COM+ 组件的桥梁。虽然基本做法类似于传统 DLL 的内部函数表格，但已经有所改进，并称之为类型库 (Type Library)。

在 7.3.3 节中，我们按照图 7-12 创建了一个类型库，并得到一个由 Delphi 自动生成的类型库文件 (.tlb)，如示例程序 7-5 所示。

示例程序 7-5 一个 Delphi 的类型库文件

```
unit DemoSvr_ TLB;
```

```
//
```

```
*****
```

```

//
// WARNING
// - - - - -
// The types declared in this file were generated from data read from a
// Type Library. If this type library is explicitly or indirectly (via
// another type library referring to this type library) re-imported, or the
// 'Refresh' command of the Type Library Editor activated while editing the
// Type Library, the contents of this file will be regenerated and all
// manual modifications will be lost.
//
* * * * *
//

// PASTLWTR : 1.2
// File generated on 03-4-9 21: 09: 18 from Type Library described below.

//
* * * * *
//
// Type Lib: E: \Delphi 面向对象编程思想\示例程序\封装\用 COM+ 封装对象\DemoSvr.tlb (1)
// LIBID: {B96FFC20-65BC-11D7-B847-001060806215}
// LCID: 0
// Helpfile:
// HelpString: DemoSvr Library
// DepndLst:
// (1) v2.0 stdole, (C: \WINDOWS\SYSTEM\stdole2.tlb)
//
* * * * *
//
{$STYPEDADDRESS OFF} // Unit must be compiled without type - checked pointers.
{$WARN SYMBOL_ PLATFORM OFF}
{$WRITEABLECONST ON}
{$VARPROPSETTER ON}
interface

uses Windows, ActiveX, Classes, Graphics, StdVCL, Variants;

//
* * * * * //
// GUIDS declared in the TypeLibrary. Following prefixes are used:
// Type Libraries : LIBID_ xxxx
// CoClasses : CLASS_ xxxx
// DISPInterfaces : DIID_ xxxx
// Non - DISP interfaces : IID_ xxxx
//
* * * * * //
const
// TypeLibrary Major and minor versions
DemoSvrMajorVersion = 1;
DemoSvrMinorVersion = 0;

LIBID_DemoSvr: TGUID = '{B96FFC20-65BC-11D7-B847-001060806215}';
IID_IDemoComObj: TGUID = '{B96FFC21-65BC-11D7-B847-001060806215}';

```

```

CLASS _ DemoComObj: TGUID = '{B96FFC23-65BC-11D7-B847-001060806215}';
type

//
*****//
// Forward declaration of types defined in TypeLibrary
//
*****//
IDemoComObj = interface;

//
*****//
// Declaration of CoClasses defined in Type Library
// (NOTE: Here we map each CoClass to its Default Interface)
//
*****//
DemoComObj = IDemoComObj;

//
*****//
// Interface: IDemoComObj
// Flags:      (256) OleAutomation
// GUID:      {B96FFC21-65BC-11D7-B847-001060806215}
//
*****//
IDemoComObj = interface (IUnknown)
    ['{B96FFC21-65BC-11D7-B847-001060806215}']
    procedure Drive; stdcall;
    procedure ride; stdcall;
end;

//
*****//
// The Class CoDemoComObj provides a Create and CreateRemote method to
// create instances of the default interface IDemoComObj exposed by
// the CoClass DemoComObj. The functions are intended to be used by
// clients wishing to automate the CoClass objects exposed by the
// server of this typelibrary.
//
*****//
CoDemoComObj = class
    class function Create; IDemoComObj;
    class function CreateRemote (const MachineName: string): IDemoComObj;
end;

implementation

uses ComObj;

class function CoDemoComObj.Create; IDemoComObj;
begin
    Result := CreateComObject (CLASS _ DemoComObj) as IDemoComObj;
end;

```

```

class function CoDemoComObj.CreateRemote (const MachineName: string): IDemoComObj;
begin
    Result := CreateRemoteComObject (MachineName, CLASS_ DemoComObj)
            as IDemoComObj;
end;

end.

```

示例程序 7-5 是一个典型的 Delphi 类型库文件，我们不妨就以它来分析。首先，我们看到 `LIBID_DemoSvr: TGUID = '{B96FFC20-65BC-11D7-B847-001060806215}'`，这里的 TGUID 刚好是 Windows 的全局唯一标识符（GUID）格式。通过搜索 Windows 的注册表（RegEdit.exe），如图 7-3 所示，我们可以发现在注册表的以下节点注册了 COM 对象的实际位置。

```

HKEY_CLASSES_ROOT \ TypeLib \ {B96FFC20-65BC-11D7-B847-001060806215}
    ↳ \ 1.0 \ 0 \ win32

```



图 7-3 在 Windows 注册表中注册的 COM 对象

这样，只要注册了 COM 对象，就不用担心应用程序找不到它。所以在主程序中调用 COM 时，不必指定 COM 的位置。

我们再看一下 COM 对象是如何创建的。注意这段代码：

```

class function CoDemoComObj.Create: IDemoComObj;
begin
    Result := CreateComObject (CLASS_ DemoComObj) as IDemoComObj;
end;

```

这里的 `CreateComObject` 是如下定义的：

```

function CreateComObject (const ClassID: TGUID): IUnknown;
begin
    OleCheck (CoCreateInstance (ClassID, nil, CLSCTX_ INPROC_ SERVER or
        CLSCTX_ LOCAL_ SERVER, IUnknown, Result));
end;

```

这说明 `CoCreateInstance` 函数利用输入的 `CLASS_ DemoComObj` 标识符（GUID）来创建 COM 对象实例。并在 `CoDemoComObj.Create` 中转型成接口类型。于是我们可以通过 COM 接口来调用 COM 内部的方法。

虽然这个过程看起来很复杂，但我们可以解决 DLL 与主程序之间的版本控制问题。由于 COM 提供了接口作为中介，而不是直接调用 COM 内部的方法（函数），因此当我们改动和调整

COM 的内部方法时，仍然可以保留原有的接口不变。这样就解决了 COM 与主程序之间版本协调的问题。另外，通过标准的 COM 接口和类型库的使用可以实现开发语言的无关性。

至于 COM+ 则是 COM 的最新发展，它包含了 COM、DCOM、MTS 等各种技术的支持。在这个示例程序 7-5 中我们还可以看到创建远程 COM 对象的方法，这实际上相当于 DCOM 的功能，不过在 Delphi 7 中已经不再分得那么细了。

```
class function CoDemoComObj.CreateRemote (const MachineName: string):  
  IDemoComObj;  
begin  
  Result := CreateRemoteComObject (MachineName, CLASS_ DemoComObj)  
    as IDemoComObj;  
end;
```

COM/COM+ 的使用特别有利于团队开发，因为这样既不用担心开发成员使用的语言差异，也不用担心某个成员对公用模块的改动而造成整个产品无法使用的困境。可以说 COM/COM+ 的使用实现了一个较高水平的物理封装境界。

7.3.2 用 DLL 封装对象

DLL (Dynamic Link Library, 动态链接库) 就目前来讲已经不再是什么新技术，读者可以在书店过时的 Delphi 书籍里随处找到讨论 DLL 编程的章节。但这些涉及 DLL 编程的书中几乎都是谈论用 DLL 来封装函数的，实际上许多程序员也是在使用 DLL 来封装函数，或面向过程的一个模块（一个函数集合）。而在这里，我只想讨论如何用 DLL 来封装对象，这可能是读者未曾有过的 DLL 使用经验，但这却是这本完全围绕面向对象编程的书中重要的部分之一，或许你能从中发现一些与众不同的实用技巧。

参见 考虑到目前关于 DLL 的现成资料很多，这里我省略了 DLL 的基本知识和编写方法，并且假设读者已经有了一定的 DLL 编程基础。如果你没有这样的基础，建议参阅本人所著的《Delphi 6 企业级解决方案及应用剖析》“DLL 编程技术”一节 (P271)。

一般来说，使用 DLL 封装对象主要有以下好处：

- 节约内存。多个程序可以使用同一个 DLL 时，该 DLL 只需加载一次，而且可以只在使用时加载，不用时销毁。
- 使程序代码实现复用。这就是说用 DLL 封装的对象可以重复使用，甚至可以让不同的程序语言调用。
- 使程序模块化、组件化。这样有利于团队开发，并且维护和更新方便。

然而，DLL 在封装对象方面却有一定的技术难度，这方面资料极少，甚至有的程序员误以为 DLL 只支持封装函数，不支持封装对象。

通过研究，我们发现 DLL 在封装对象上主要的限制在于：

- 调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法。
- DLL 封装对象的实例只能在 DLL 中创建。
- 在 DLL 和调用 DLL 的应用程序中都需要对封装的对象及其被调用的方法进行声明。

下面我先通过一个简单的例子来演示如何使用 DLL 封装对象，并在应用程序中调用该对

象。然后再讨论相关的技术细节。

读者一定还记得我们在示例程序 3-13 中演示了车的继承关系和合成关系。这个程序由逻辑单元的 Demo 和界面单元 frmDemo 组成。我们现在就用 DLL 封装 Demo 单元的所有对象，并在 frmDemo 单元实现调用。读者可以通过这个具体的例子来学习如何使用 DLL 封装对象。

打开示例程序 3-13 的项目文件 ObjDemo.dpr，如图 7-4 所示，在项目管理器（Project Manager）中，右击 ProjectGroup1，然后在弹出菜单中选择 Add New Project 菜单项。此时弹出如图 7-5 所示的 New Items 对话框。选择 DLL Wizard，Delphi 的 DLL 向导将创建一个 DLL 项目。我们将该项目重新命名为 DemoSvr，并保存在项目组同一目录下。

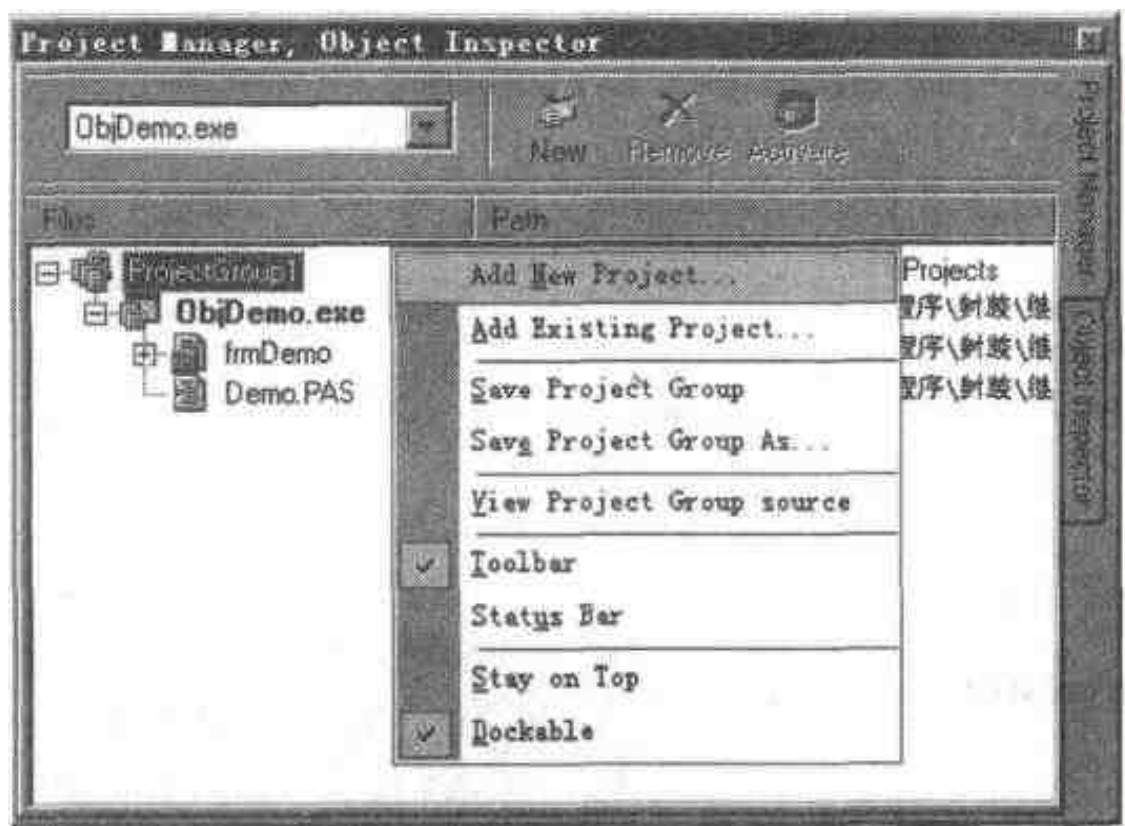


图 7-4 鼠标右击 ProjectGroup1，在弹出菜单中选择 Add New Project 菜单项

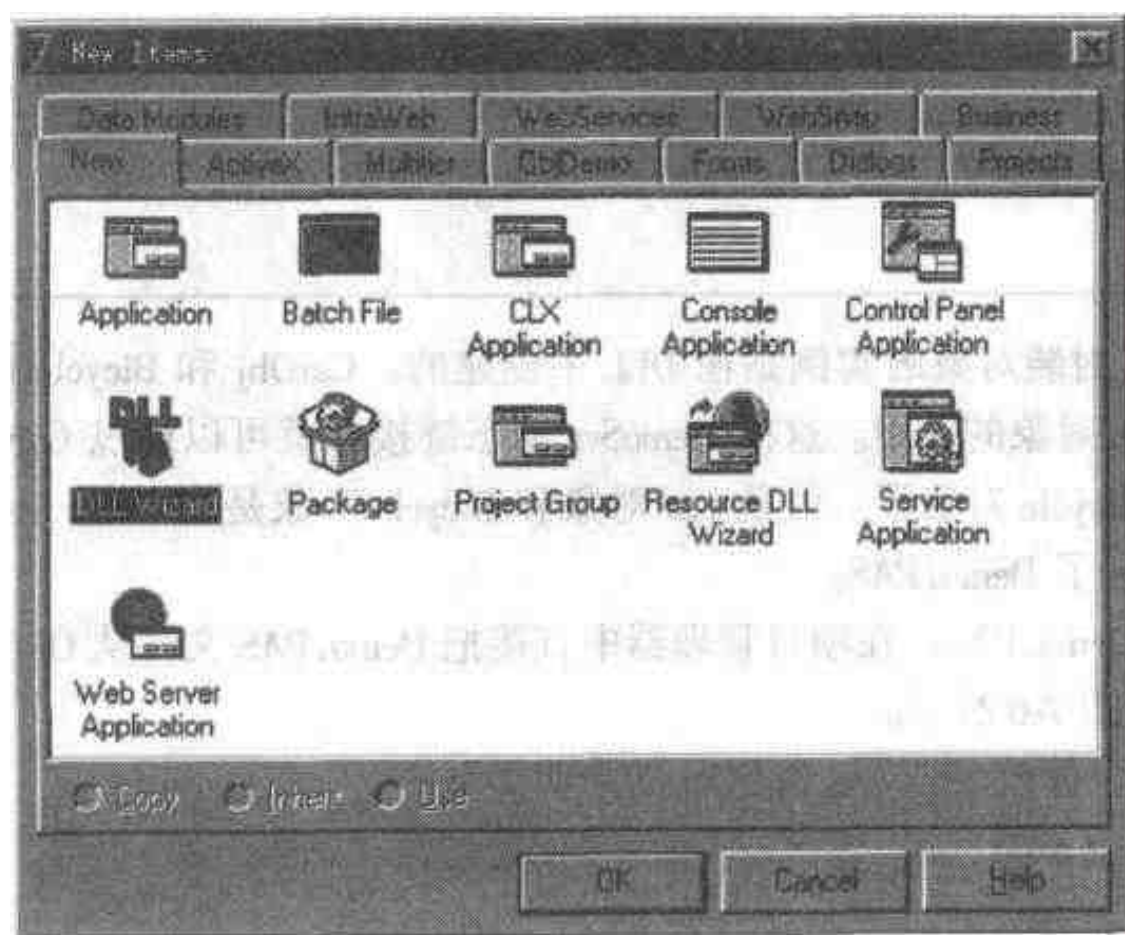


图 7-5 在 New Items 对话框中选择 DLL Wizard

修改 DemoSvr 中的代码如示例程序 7-6 所示。

示例程序 7-6 动态链接库 DemoSvr 的主程序

```
library DemoSvr;  
  
{Important note about DLL memory management: ShareMem must be the  
first unit in your library's USES clause AND your project's (select  
Project-View Source) USES clause if your DLL exports any procedures or  
functions that pass strings as parameters or function results. This  
applies to all strings passed to and from your DLL—even those that  
are nested in records and classes. ShareMem is the interface unit to  
the BORLNDMM.DLL shared memory manager, which must be deployed along  
with your DLL. To avoid using BORLNDMM.DLL, pass string information  
using PChar or ShortString parameters.}  
  
uses  
    ShareMem,  
    SysUtils,  
    Classes,  
    Demo in 'Demo.PAS';  
  
{$R *.res}  
  
function CarObj: TCar;  
begin  
    Result: = TCar.create;  
end;  
  
function BicycleObj: TBicycle;  
begin  
    Result: = TBicycle.create;  
end;  
  
exports  
    CarObj,  
    BicycleObj;  
end.
```

由此可见, DLL 封装对象的实例是在 DLL 中创建的, CarObj 和 BicycleObj 函数创建并输出了 Car 对象和 Bicycle 对象的引用。这样 DemoSvr 动态链接库就可以通过 CarObj 和 BicycleObj 函数输出 Car 对象和 Bicycle 对象了。但是 Car 对象和 Bicycle 对象是在 Demo.PAS 文件中声明和实现的, 所以这里 uses 了 Demo.PAS。

为了能够使用 Demo.PAS, 在项目管理器中直接把 Demo.PAS 文件从 ObjDemo 项目中拖放到 DemoSvr 项目中, 如图 7-6 所示。

打开 Demo.PAS, 修改 TBicycle 和 TCar 的声明如下:

```
TBicycle = class (TVehicle)  
public  
    constructor create;  
    destructor Destory;
```

```
procedure ride; virtual;  
end;  
  
TCar = class (TVehicle)  
protected  
    FEngine: TEngine;  
public  
    constructor create;  
    destructor Destory;  
    procedure drive; virtual;  
end;
```

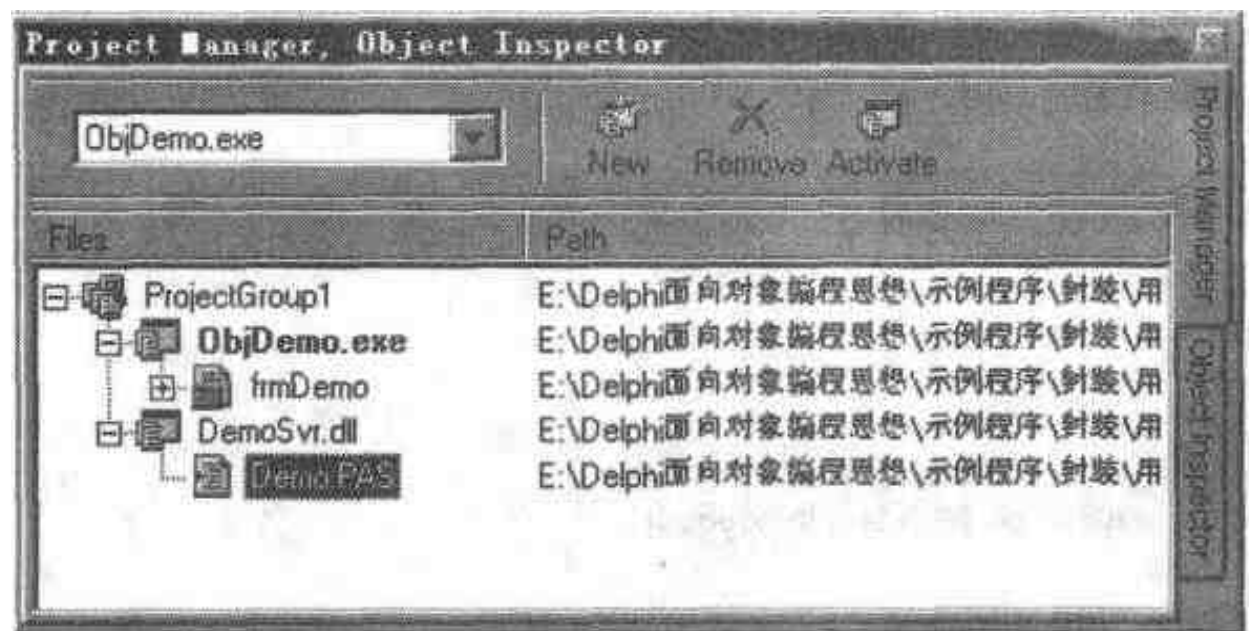


图 7-6 将 Demo.PAS 文件从 ObjDemo 项目中拖放到 DemoSvr 项目中

请注意，这里我把应用程序中需要调用的对象方法 ride 和 drive 改成了虚方法。显然，这么做不是为了让 TBicycle 和 TCar 的派生类来覆盖 ride 和 drive 方法。这是因为编译连接应用程序时，编译器无法知道（也无需知道）对象在 DLL 中的方法是如何实现的，这就意味着对于应用程序来说要使用动态绑定（晚绑定）技术，所以调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法。前面我们讲过，虚方法的动态绑定技术是把虚方法的入口放到虚方法表 VMT 中。VMT 是一块包含对象方法指针的内存区，通过 VMT 调用程序可以得到虚方法的指针。如果我们不把 ride 和 drive 声明为虚方法，VMT 中就不会有这些方法的入口指针，因此调用程序也就无法得到这个方法入口的指针。

接下来回到 frmDemo 单元，在调用 DLL 的应用程序中同步声明需要调用的对象及其被调用的方法。这里除了将 ride 和 drive 声明为虚方法外，还要声明为抽象方法。因为 frmDemo 单元不提供 ride 和 drive 方法的实现，如果不把它们声明为抽象方法则在编译时就无法通过。应用程序 frmDemo 单元的完整代码如示例程序 7-7 所示。

示例程序 7-7 调用 DLL 对象的应用程序

```
unit frmDemo;  
  
interface  
  
uses  
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
```

```

    Dialogs, StdCtrls;

type

// - - 这里声明需要利用的 DLL 中对象的方法 -

    TVehicle = class (TObject);
    TCar = class (TVehicle)
    public
        procedure drive; virtual; abstract;
    end;
    TBicycle = class (TVehicle)
    public
        procedure ride; virtual; abstract;
    end;

// - - - - -

    TForm1 = class (TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button2Click (Sender: TObject);
    procedure Button1Click (Sender: TObject);
    private
        {Private declarations}
    public
        {Public declarations}
    end;

var
    Form1: TForm1;
// - - - 这里导入 DLL 文件及其函数 - - -
    function CarObj: TCar; external 'DemoSvr.dll';
    function BicycleObj: TBicycle; external 'DemoSvr.dll';

implementation

{$R *.dfm}

procedure TForm1.Button2Click (Sender: TObject);
var MyCar: TCar;
begin
    MyCar := CarObj;
    if MyCar = nil then exit;
    try
        MyCar.drive;
    finally
        MyCar.Free;
    end;
end;

procedure TForm1.Button1Click (Sender: TObject);
var Bicycle: TBicycle;
begin

```

```
Bicycle := BicycleObj;  
try  
    Bicycle.ride;  
finally  
    Bicycle.Free;  
end;  
end;  
  
end.
```

最后，选择 Build All Projects 菜单项，编译和连接所有的项目，如图 7-7 所示，我们就得到了需要的应用程序可执行文件以及 DLL。运行测试，可以看到这个程序实现了和原先一样的功能。

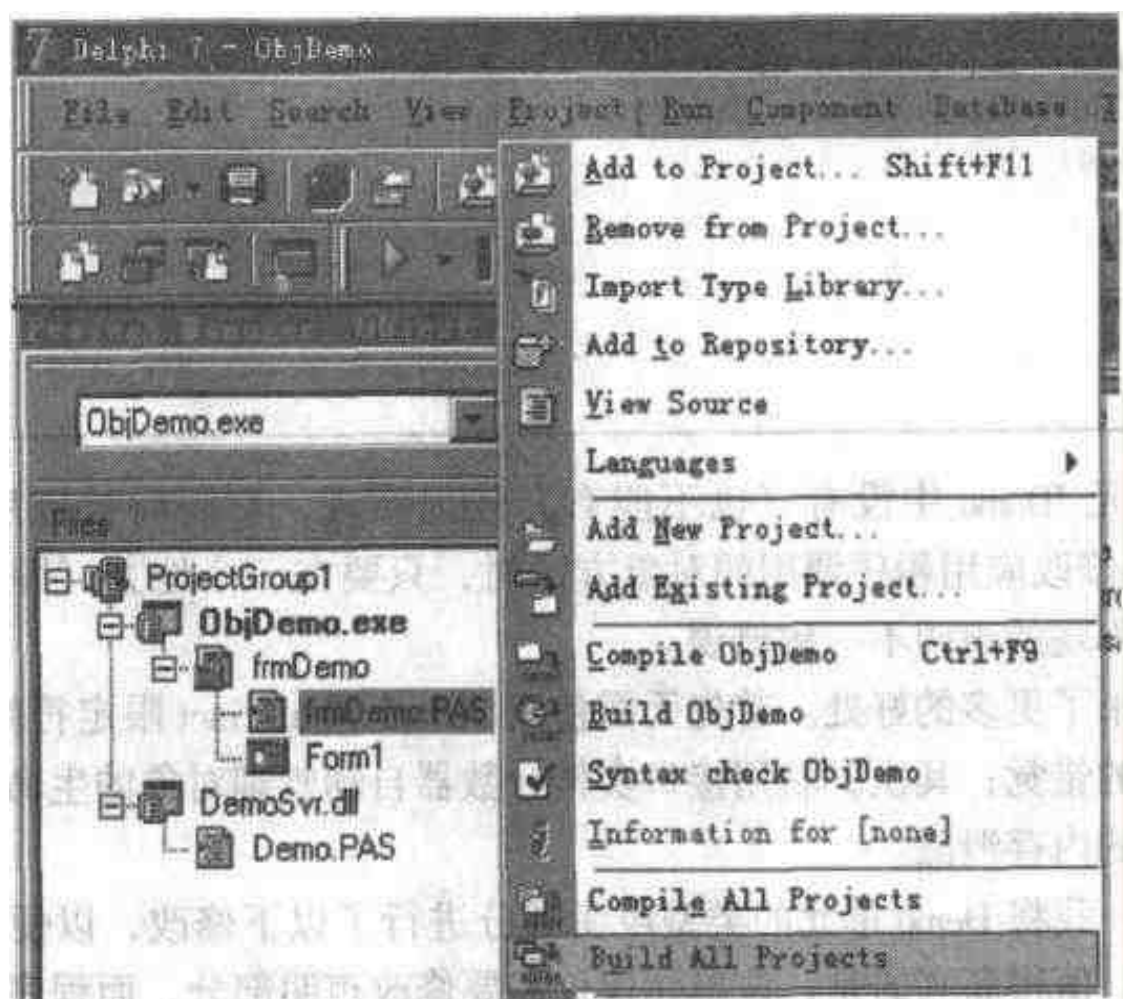


图 7-7 选择 Build All Projects 菜单项，编译和连接所有的项目

但是我对这样的 DLL 封装对象实现不是太满意。因为在 DLL 和应用程序中都需要声明封装的对象，还要使用好 virtual 和 abstract 限定符（很容易造成阅读程序理解上的错觉）。如果一旦对象发生变化，就需要分别在两边修改对象声明，以保持同步，稍有不慎就会出错。对此，Steve Teixeira 在《Delphi 6 开发人员指南》（机械工业出版社 2003 年翻译出版，相关内容参见该书 209 页）一书中提出了使用头文件的方法，并通过加上编译指令来控制 DLL 和应用程序分别读到不同的头文件内容。这个方法虽然可以通过只修改头文件来保持声明的同步，但编译指令和头文件使得阅读程序更加困难。

在这里我有一个更好的方法与读者分享，这就是使用接口的方法。

前面我们分析过，调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法，理解这一点是实现 DLL 封装和使用对象的关键。而 Delphi 接口技术为我们提供了一个最佳选择。

为此我们创建一个接口单元 IDemo，分别声明 ICar 和 IBicycle 接口，接口方法分别是应用

程序要用到的 Drive 和 Ride。完整代码如示例程序 7-8 所示。

示例程序 7-8 接口单元 IDemo 的代码

```

unit IDemo;

interface

type
  ICar = interface (IInterface)
    ['{ED52E264-6683-11D7-B847-001060806215}']
    procedure Drive;
  end;

  IBicycle = interface (IInterface)
    ['{ED52E264-6683-11D7-B847-001060806216}']
    procedure Ride;
  end;

implementation

end.

```

注意，接口单元 IDemo 中没有（也不能有）任何实现，它同时被应用程序和 DLL 所用 (Use)，这样当需要修改应用程序调用的对象方法时，只要在一个地方（即该接口单元）修改即可，避免了可能出现的声明不一致错误。

使用接口还带来了更多的好处。首先无需使用 `virtual` 和 `abstract` 限定符修改对象方法声明，避免了程序阅读上的错觉；其次，利用接口实例计数器自动管理对象的生命期，避免了程序员遗忘销毁对象造成的内存泄漏。

为了使用接口，我将 Demo 单元的类型声明部分进行了以下修改，以便 TBicycle 和 TCar 类能够实现接口方法。值得高兴的是，该单元仅仅需要修改声明部分，而程序实现部分根本不需要做任何改动。

```

unit Demo;

interface

uses
  SysUtils, Windows, Messages, Classes, Dialogs, IDemo;

type
  TVehicle = class (TInterfacedObject)
  protected
    FColor: string;
    FMake: string;
    FTopSpeed: Integer;
    FWheel: TWheel;
    FWheels: TList;
    procedure SlowDown;
    procedure SpeedUp;
  end;

```

```

    procedure Start;
    procedure Stop;
end;

TBicycle = class (TVehicle, IBicycle)
public
    constructor create;
    destructor Destory;
    procedure ride;
end;

TCar = class (TVehicle, ICar)
protected
    FEngine: TEngine;
public
    constructor create;
    destructor Destory;
    procedure drive;
end;

```

最后检查一下项目管理器，确保在应用程序项目和 DLL 项目中都添加了 IDemo 单元，如图 7-8 所示。

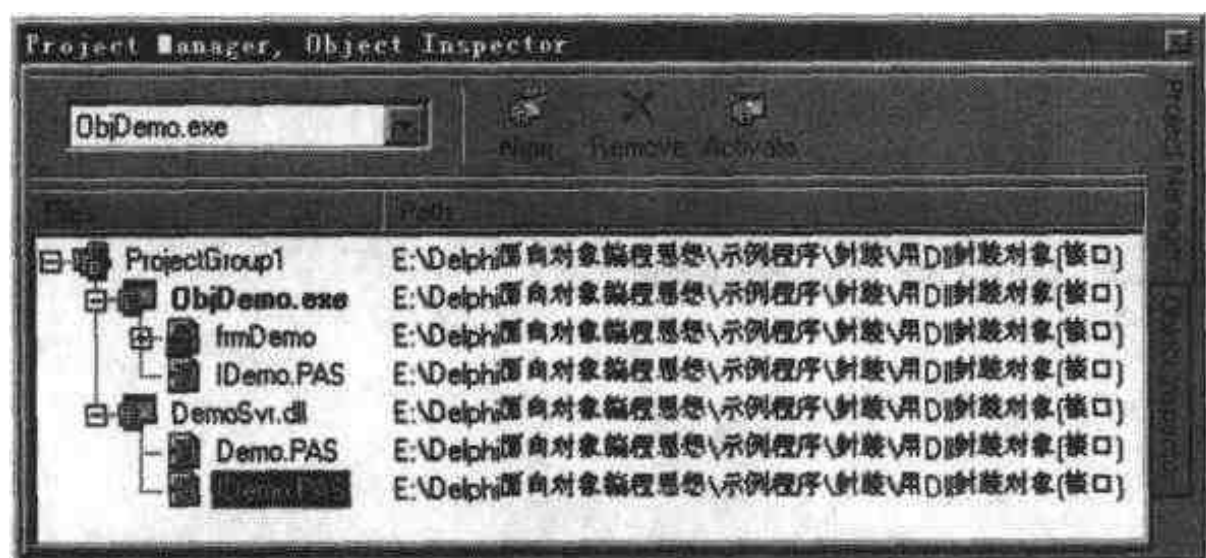


图 7-8 确保在应用程序项目和 DLL 项目中都添加了接口单元 IDemo

在示例程序 7-9 中，改动不是很多。这里 Use 了 IDemo 单元，而没有额外的声明。实现部分通过接口调用了 DLL 中的接口方法，也可以说是对象方法。运行示例程序 7-9 和运行示例程序 7-7，实现的功能完全一样。

示例程序 7-9 使用接口技术调用 DLL 对象的应用程序

```

unit frmDemo;

interface

uses
    Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls, IDemo; //在这里 Use IDemo 单元

type

```

```

TForm1 = class (TForm)
  Button1: TButton;
  Button2: TButton;
  procedure Button2Click (Sender: TObject);
  procedure Button1Click (Sender: TObject);
private
  {Private declarations}
public
  {Public declarations}
end;

var
  Form1: TForm1;
  function CarObj: ICar; external 'DemoSvr.dll';
  function BicycleObj: IBicycle; external 'DemoSvr.dll';

implementation

{$R *.dfm}

procedure TForm1.Button2Click (Sender: TObject);
var MyCar: ICar;
begin
  MyCar := CarObj;
  MyCar.drive;
  MyCar := nil;
end;

procedure TForm1.Button1Click (Sender: TObject);
var Bicycle: IBicycle;
begin
  Bicycle := BicycleObj;
  Bicycle.ride;
  Bicycle := nil;
end;

end.

```

7.3.3 用 COM/COM+ 封装对象

初学 COM 编程的朋友通常会被对象、接口与动态链接库 (DLL) 三者之间的关系弄得一头雾水! COM 的实现是以 DLL 为载体, COM 通过接口与客户程序通信, 接口必须委托给类实现。前面我们谈论了将类封装到动态链接库中, 并通过 DLL 导出对象引用, 其中还讲到了接口的妙用, 这实际上就已经包含了 COM 的影子。

COM (Component Object Model) 是微软提出的组件对象模型, 用 COM 来封装对象可以使对象具有通用、可扩展、易维护等组件特性。COM+ 是 COM 的最新版本, 作为 Windows 2000 和 Windows XP 的标准部分一起推出。COM+ 没有什么特别新奇之处, 它只是对 COM 进行了改进, 并集成了微软的 DCOM (分布式 COM)、MTS (微软事务服务器) 和 MSMQ (微软消息队列服务器) 等一大堆相关的技术和产品。

这里并不准备深入研究 COM/COM+ 编程, 而只想通过一个实例来剖析如何使用 COM/COM+ 封装对象。我使用的仍然是与前面相似的例子, 以便读者进行比较。

打开示例程序 3-13 的项目文件 ObjDemo.dpr, 如图 7-4 所示, 在项目管理器 (Project Manager) 中, 鼠标右击 ProjectGroup1, 然后在弹出菜单中选择 Add New Project 菜单项。此时弹出如图 7-9 所示的 New Items 对话框。选择 ActiveX Library 后, Delphi 将创建一个 COM/COM+ 项目。我们将该项目重新命名为 DemoSvr, 并保存在项目组同一目录下, 如图 7-13 所示。

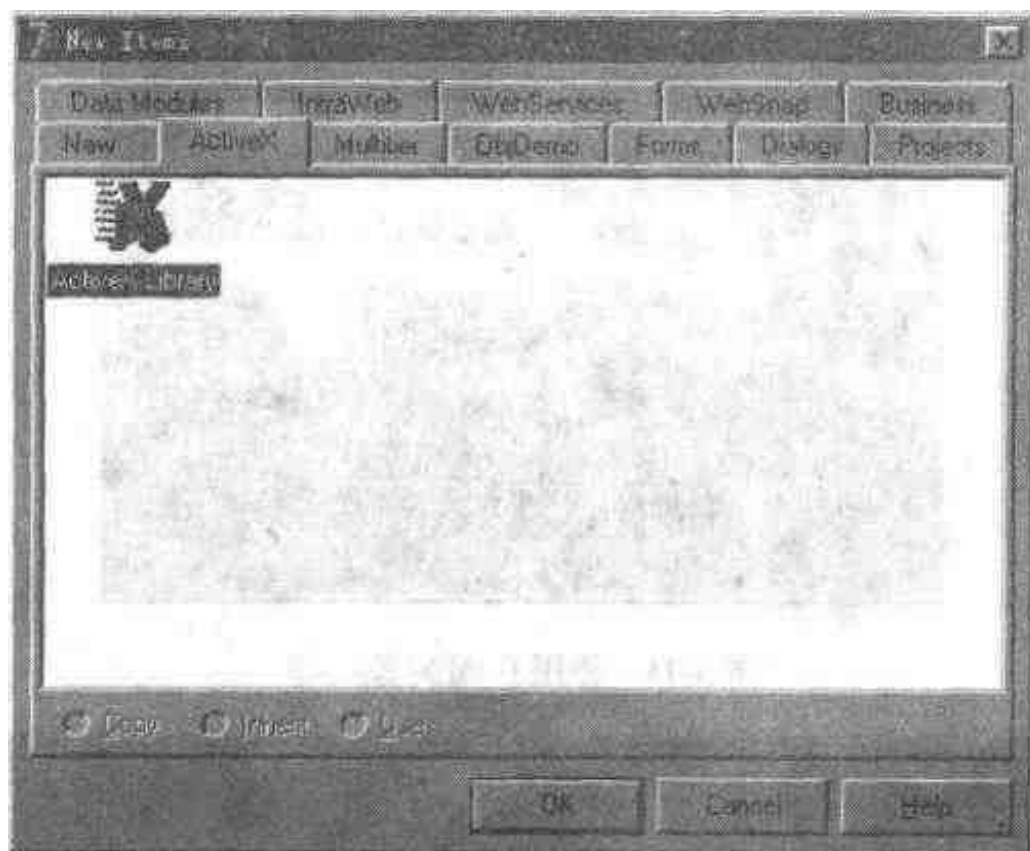


图 7-9 在 New Items 对话框中选择 ActiveX Library

选择菜单 File|New|Other, 打开如图 7-10 所示的 New Items 对话框。选择 COM Object, 此时弹出 COM 对象向导, 如图 7-11 所示, 输入类名称, 点击 OK 按钮。并在接着出现的类型库

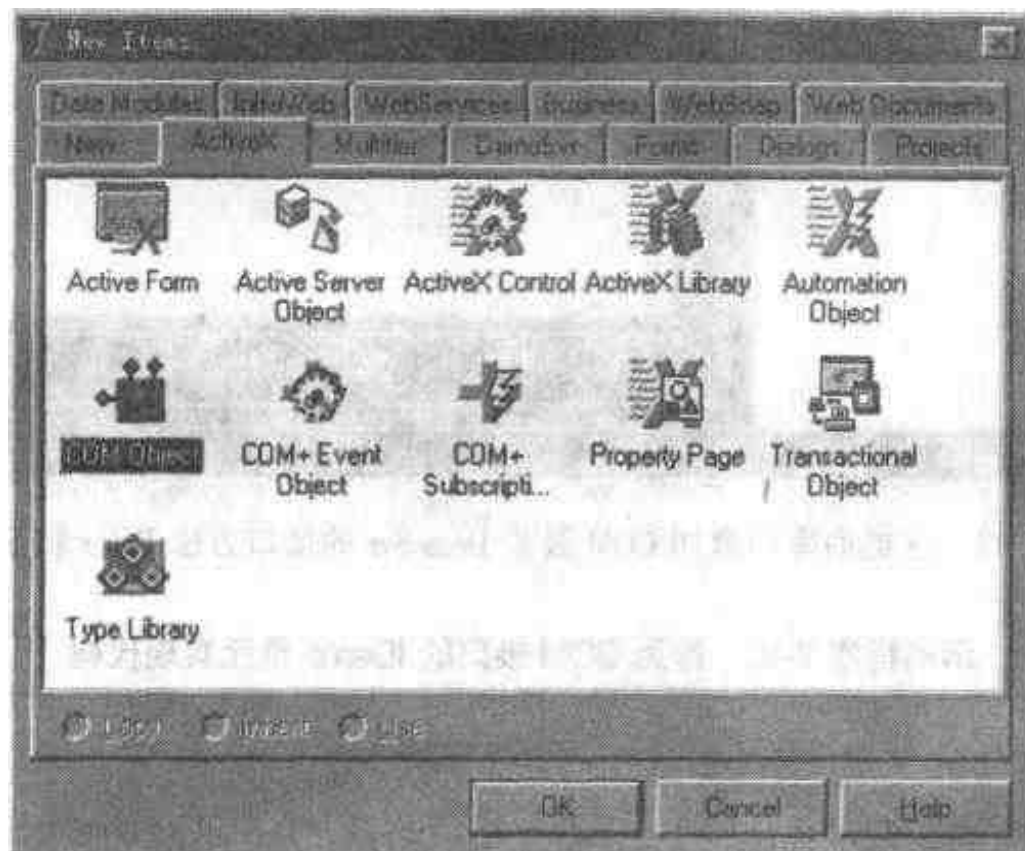



图 7-10 在 New Items 对话框中选择 COM Object

DemoSvr 中添加 COM 服务的 IDemoComObj 接口方法 Drive 和 Ride，如图 7-12 所示。这两个简单的方法没有参数。记住点击工具栏上的刷新实现按钮，此时向导生成的 COM 单元中会出现对应的代码段，供你使用。我们填写完成这些代码段，如示例程序 7-10 所示。并将其保存为 IDemo 单元文件。

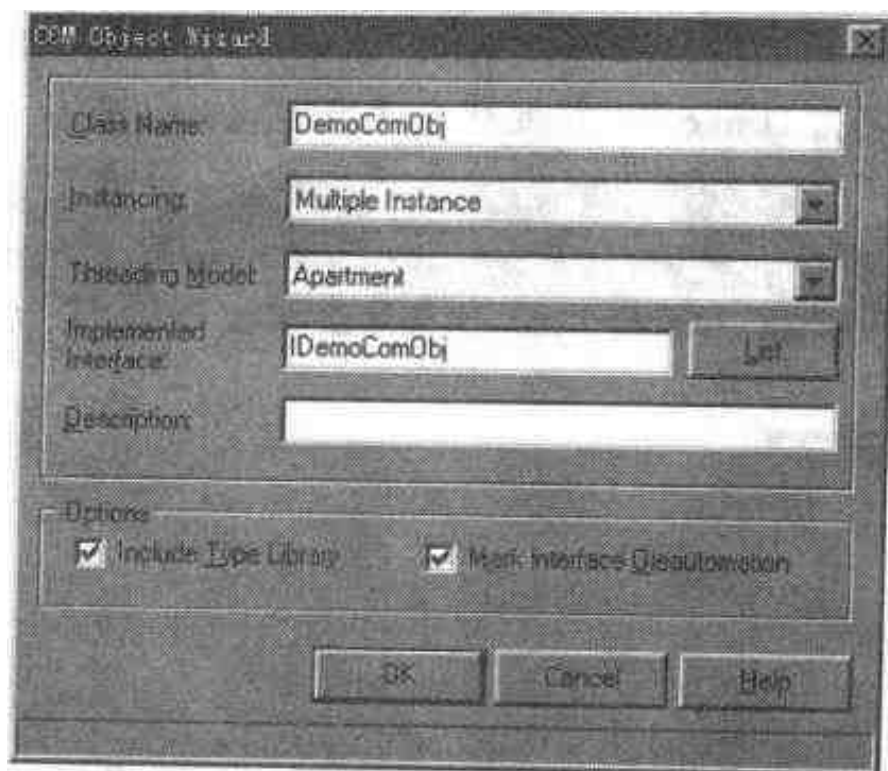


图 7-11 使用 COM 对象向导

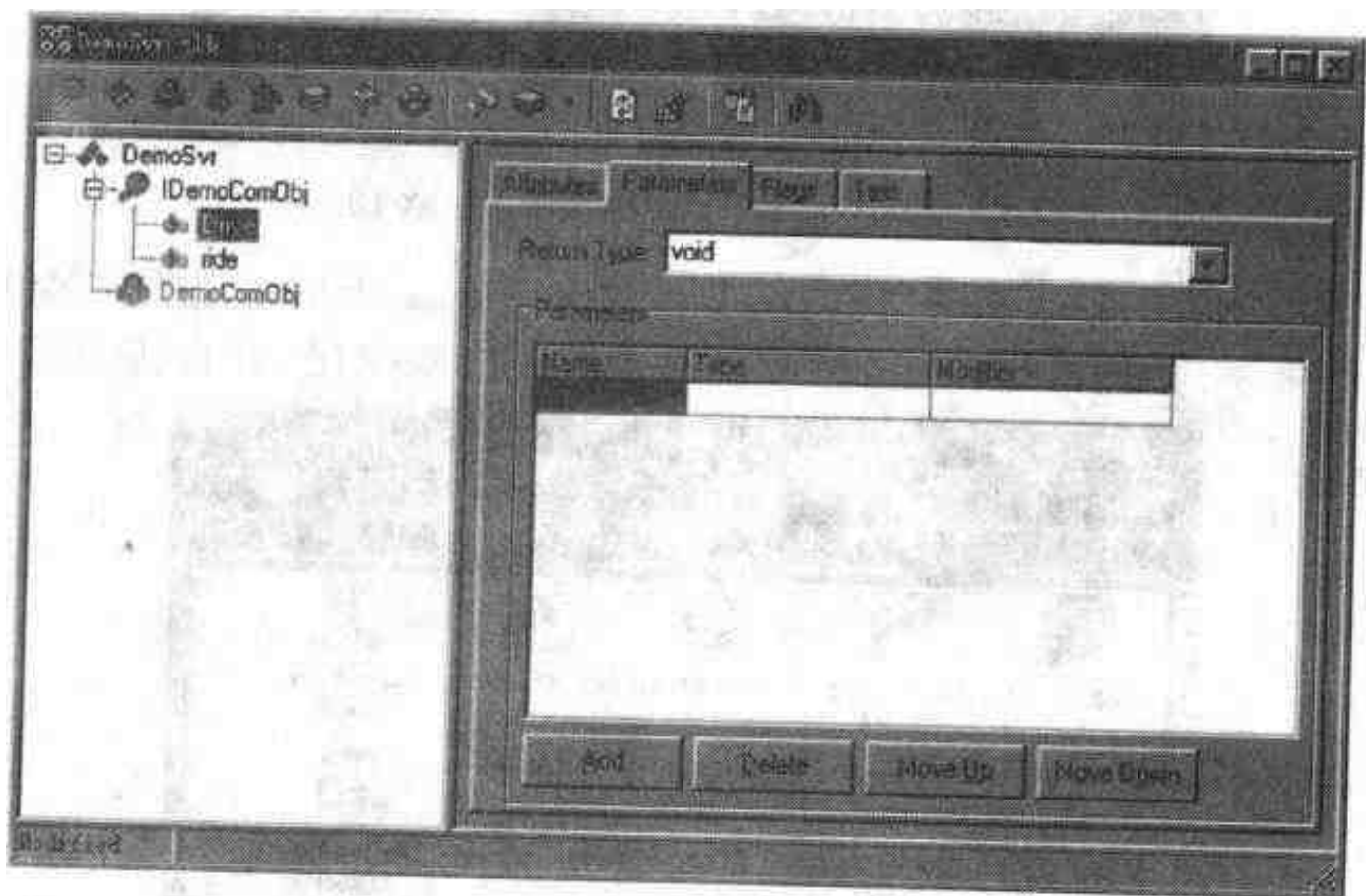


图 7-12 在类型库中添加 COM 服务 DemoSvr 的接口方法 Drive 和 Ride

示例程序 7-10 作为 COM 接口的 IDemo 单元实现代码

```
unit IDemo;

{$WARN SYMBOL_PLATFORM OFF}

interface
```

```
uses
  Windows, ActiveX, Classes, ComObj, DemoSvr _ TLB, StdVcl;

type
  TDemoComObj = class (TTypedComObject, IDemoComObj)
  protected
    procedure Drive; stdcall;
    procedure ride; stdcall;
  end;

implementation

uses ComServ, Demo;

procedure TDemoComObj.Drive;
var MyCar: TCar;
begin
  MyCar := TCar.create;
  try
    MyCar.drive;
  finally
    MyCar.Free;
  end;
end;

procedure TDemoComObj.ride;
var Bicycle: TBicycle;
begin
  Bicycle := TBicycle.create;
  try
    Bicycle.ride;
  finally
    Bicycle.Free;
  end;
end;

initialization
  TTypedComObjectFactory.Create (ComServer, TDemoComObj, Class _ DemoComObj,
    ciMultiInstance, tmApartment);
end.
```

在示例程序 7-10 中，我们看到 TDemoComObj 类继承了 IDemoComObj 接口，并实现了 IDemoComObj 的接口方法 Drive 和 Ride。

最后项目 DemoSvr 中的文件组成如图 7-13 所示。这里我们和前面一样需要将 Demo 单元文件加到项目 DemoSvr 中，但不同的是 Demo 单元中的代码无需做任何修改。这也是用 COM 封装的优势所在！

对于调用 COM/COM+ 的应用程序来说，首先要 use 类型库文件 DemoSvr _ TLB，然后创建 COM/COM+ 对象。示例程序 7-11 显示了应用程序 frmDemo 单元的完整代码，我们会发现，这和示例程序 7-9 使用接口技术调用 DLL 对象的应用程序有点相似。

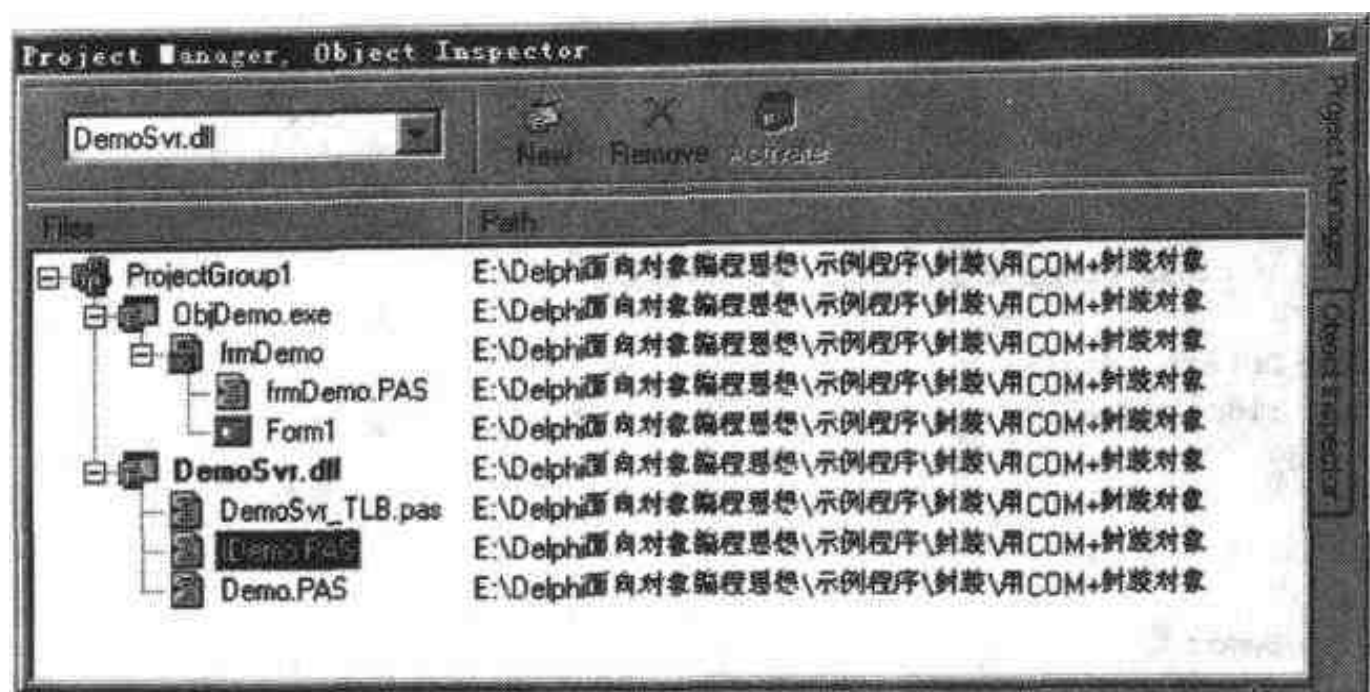


图 7-13 界面程序 ObjDemo.exe 和 COM 服务 DemoSvr.dll 的项目文件组成

示例程序 7-11 调用 COM/COM+ 封装对象的应用程序

```

unit frmDemo;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class (TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button2Click (Sender: TObject);
    procedure Button1Click (Sender: TObject);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation

uses DemoSvr _ TLB;

{$R *.dfm}

procedure TForm1.Button2Click (Sender: TObject);
var MyCar: IDemoComObj;
begin
  MyCar := CoDemoComObj.Create;
  MyCar.drive;
end;

```

```

    MyCar: = nil;
end;

procedure TForm1.Button1Click (Sender: TObject);
var Bicycle: IDemoComObj;
begin
    Bicycle: = CoDemoComObj.Create;
    Bicycle.ride;
    Bicycle: = nil;
end;

end.

```

调试运行该程序，如果出现图 7-14 所示的错误提示，就说明由于没有注册 COM 组件，导致了“类没有注册”异常（EOleSysError）。如图 7-15 所示，选择 Run|Register ActiveX Server 菜单项注册 COM 组件（实际上就是一个 ActiveX 服务）。再运行程序，我们得到了和以前一样的结果，此时封装成功。查看该 COM 组件的物理文件，发现仍然是一个动态链接库 DemoSvr.dll。



图 7-14 由于没有注册 COM 组件，导致了“类没有注册”异常（EOleSysError）

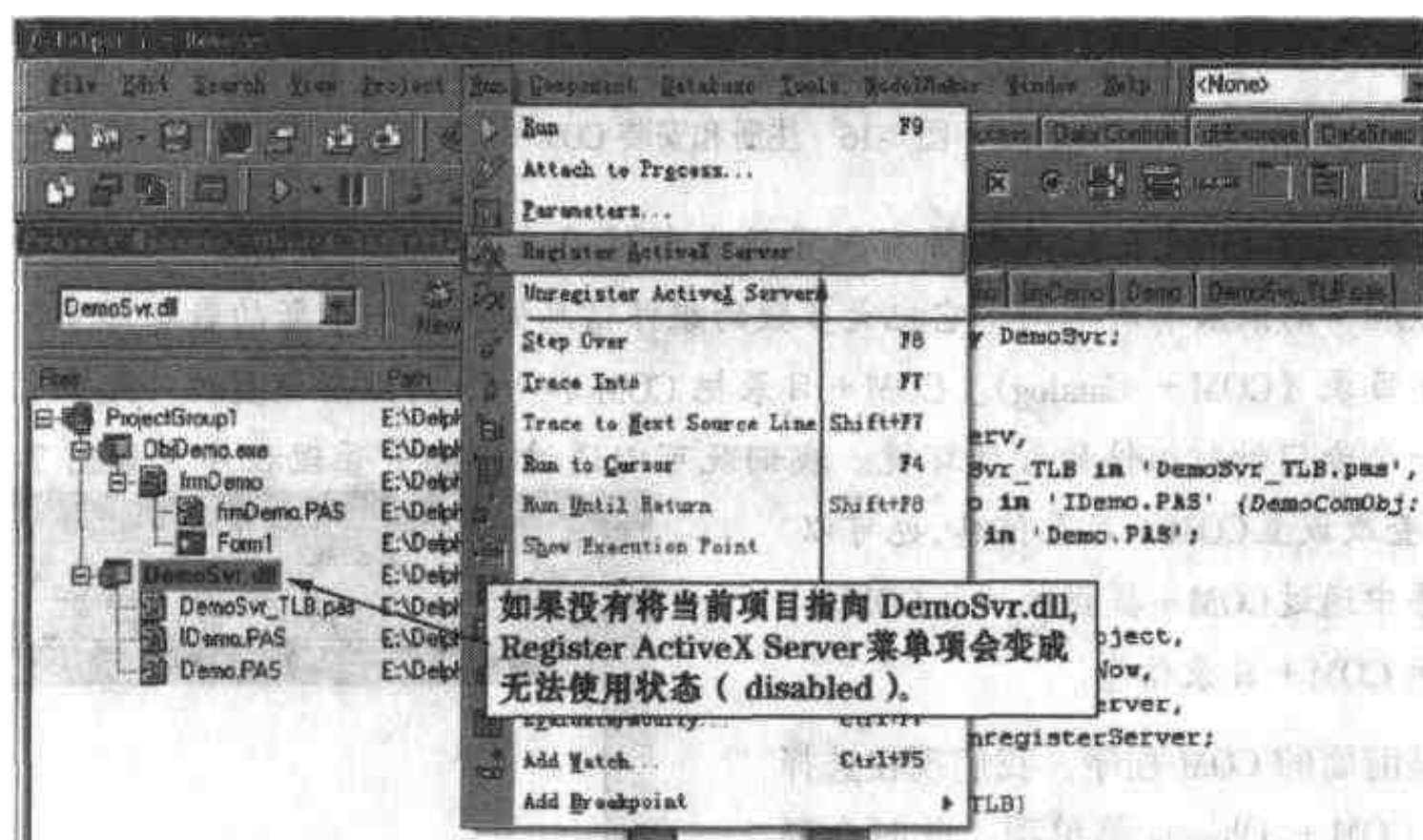


图 7-15 注册 ActiveX Server

看到这里读者可能会有疑问：前面讲的都是用 COM 封装对象，可是你却总是说 COM/COM + 封装对象，这一样吗？

其实 COM+ 的基本结构并不复杂,简单说起来,它把 COM、DCOM 和 MTS 的编程模型结合起来,同时又增加了一些新的特性。对于 Delphi 来说,创建一个简单的不需要事务的 COM+ 组件和创建一个 COM 组件几乎没有什么区别。惟一的区别在于它们的注册方式。

注意到图 7-15 中 Delphi 的开发环境是 Windows 98,由于 Windows 98 不支持 COM+,所以在菜单上没有注册和安装 COM+ 对象的选项。不同的是在支持 COM+ 的 Windows 版本中 (Windows 2000/Me/XP), Delphi 菜单上会出现 Run|Install COM+ Objects 菜单项,用于注册和安装 COM+ 组件,如图 7-16 所示。所以,我们不必修改前面的 COM 程序,只需选择不同的菜单就可以实现 COM/COM+ 封装对象。

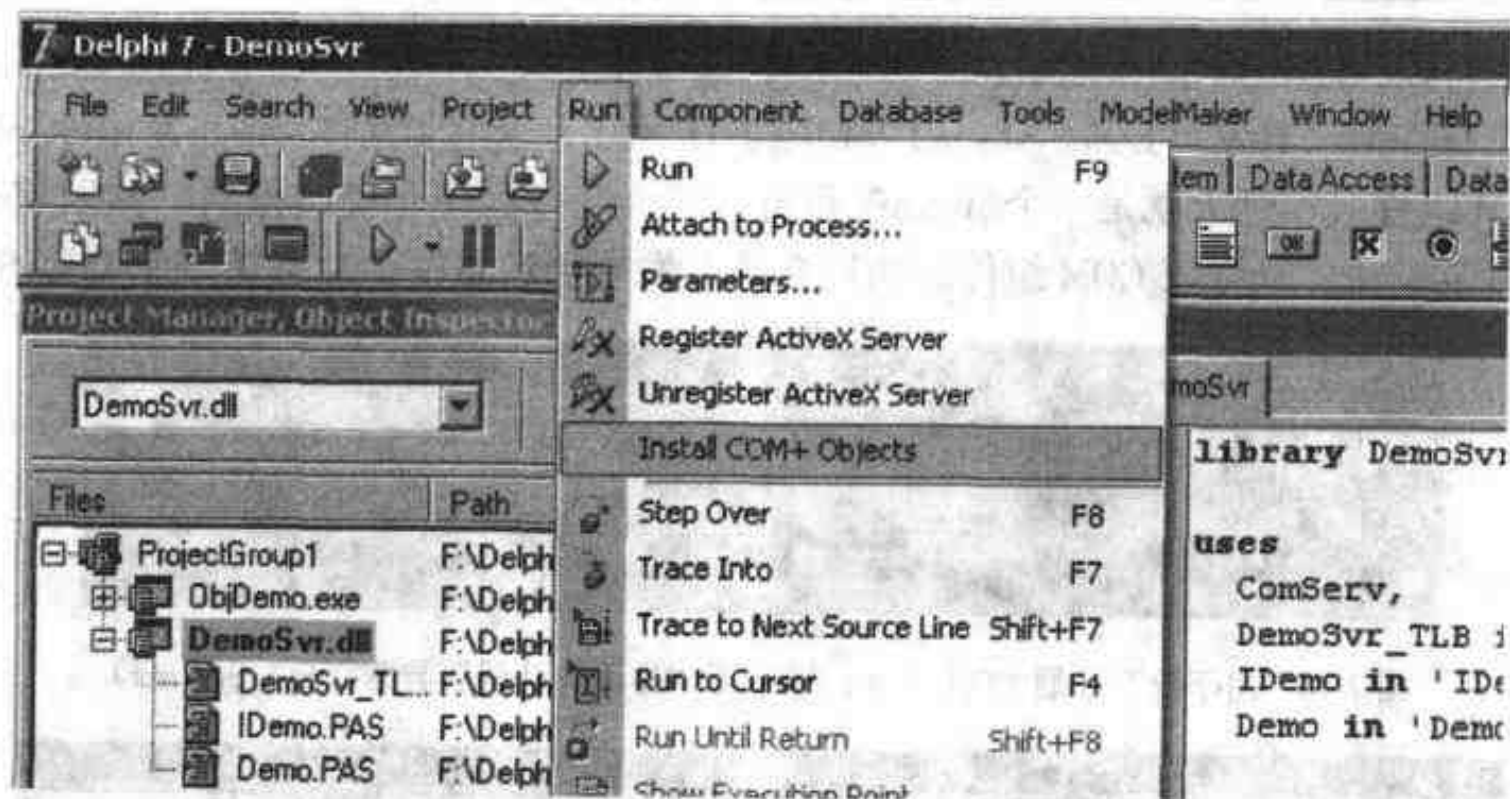


图 7-16 注册和安装 COM+

注意 COM 和 MTS 把组件的所有配置信息都保存在 Windows 的系统注册表中,然而,COM+ 的做法有所不同,它把大多数的组件信息保存在一个新的数据库中,称为 COM+ 目录 (COM+ Catalog)。COM+ 目录把 COM 和 MTS 的注册模型统一起来,并提供了一个专门针对组件的管理环境。我们既可以通过 COM+ 管理程序 (如图 7-19 所示)检查或设置 COM+ 目录信息,也可以在程序中通过 COM+ 提供的一组 COM 接口访问 COM+ 目录信息。

仍然是前面的 COM 程序,我们现在选择 Run|Install COM+ Objects 菜单项,此时会弹出一个 Install COM+ Objects 对话框,如图 7-17 所示。这是 Windows 操作系统要求我们把这个 COM+ 对象安装到一个 COM+ 应用程序中。如果你打算为这个 COM+ 对象新建一个

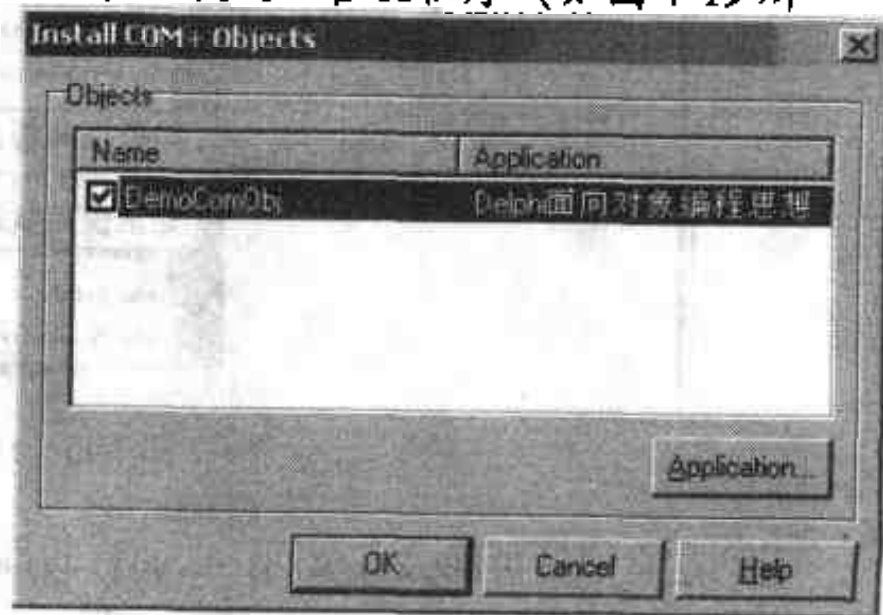


图 7-17 Install COM+ Objects 对话框

COM+ 应用程序，可以点击 Application 按钮，在如图 7-18 所示的对话框中进行设置。



图 7-18 为 COM+ 对象新建一个 COM+ 应用程序

打开 Windows 的组件服务，如图 7-19 所示，我们可以看到已经安装好的 COM+ 组件以及它的接口方法。运行应用程序，点击“驾驶汽车”按钮，我们在如图 7-20 所示的 Windows 的组件服务窗口中可以看到激活的 COM+ 组件正在旋转，说明我们用 COM+ 封装的对象在正常工作。

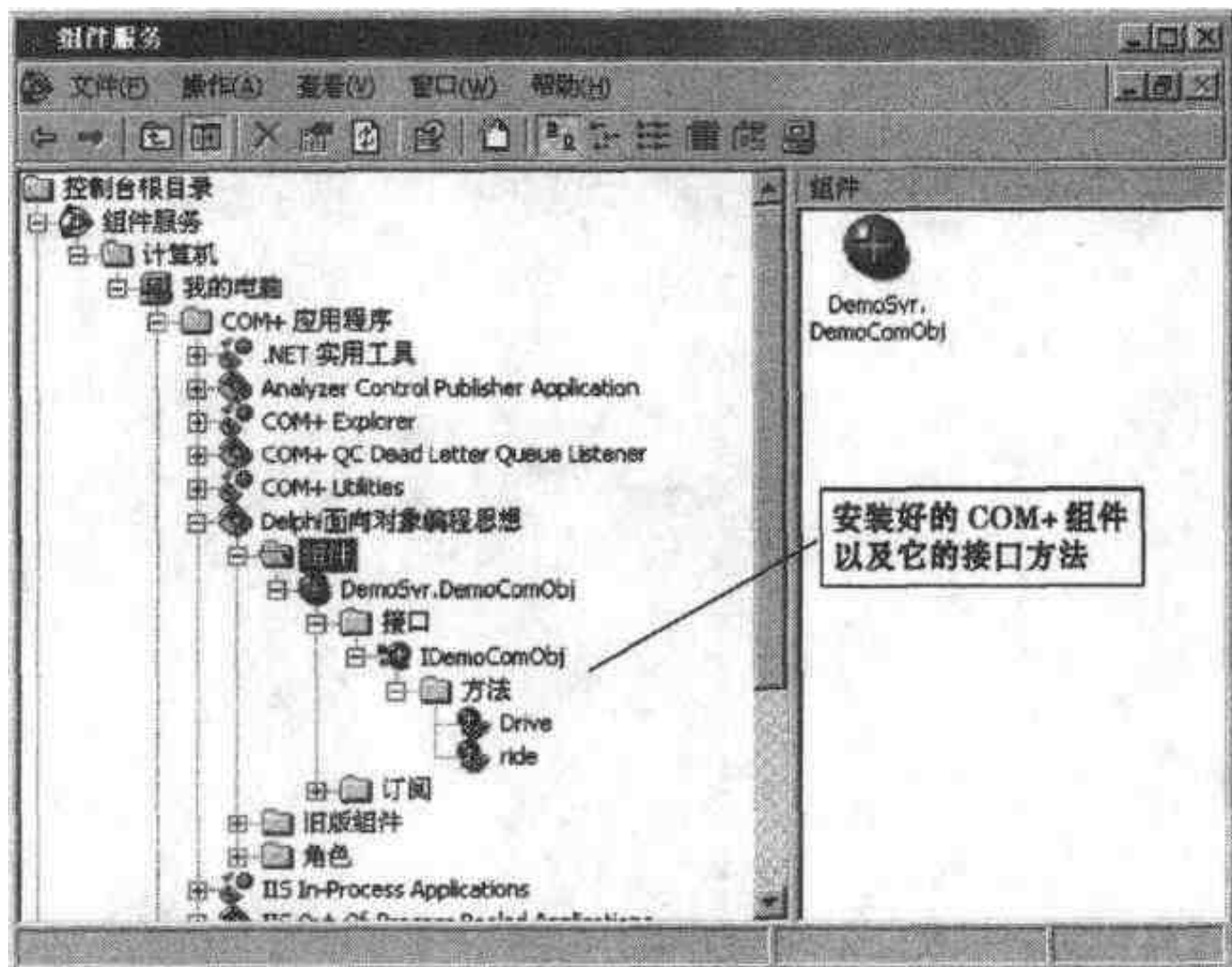


图 7-19 在“组件服务”窗口可以看到安装好的 COM+ 组件以及它的接口方法

读者可能已经感觉到，通过前面的几个例子，我们可以进一步理解对象、接口、DLL 和 COM 之间的微妙关系，特别是深刻体会对象、接口和 DLL 三者相互配合的强大作用，对深入学习 COM/COM+ 会起到事半功倍的效果。

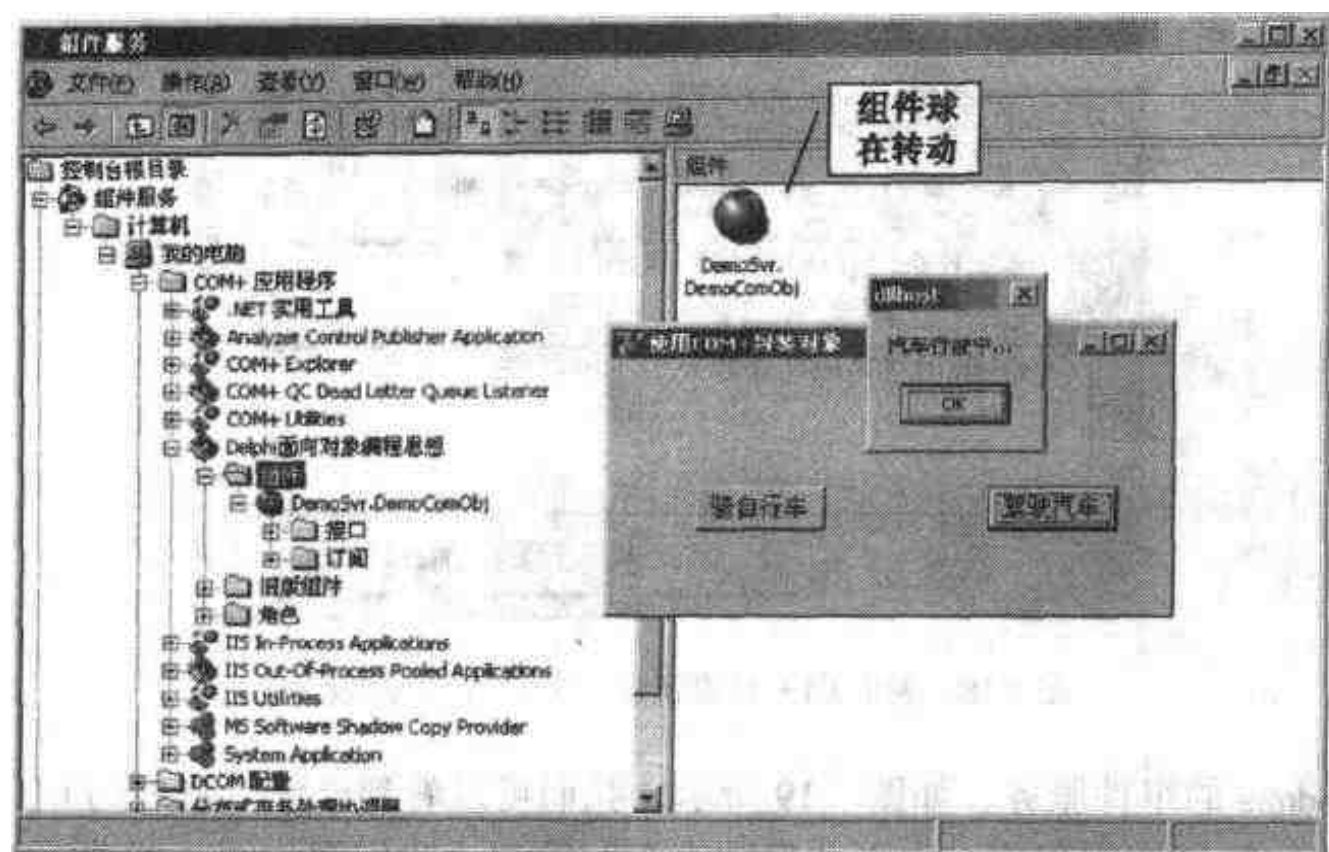


图 7-20 运行程序时，在“组件服务”窗口可以看到 COM+ 组件被激活（图中组件球在转动）

第 8 章 实现界面和业务的分离

8.1 关于界面和业务的分离

界面 (GUI) 和业务 (business) 的分离是开发可维护、易扩展、长寿命的应用系统的关键,也是实现多层分布式系统的必经之路。面向对象编程技术为开发界面和业务分离的系统提供了行之有效的途径。

8.1.1 从封装到界面和业务分离

前面我们讨论了代码的封装技术。在逻辑上,代码的封装是通过类来实现的,类通过属性封装了对象的数据,类通过方法封装了对象的行为。在物理上,代码的封装是通过不同结构和形式的文件来实现的。对于 Windows 应用程序而言,代码程序可以封装在可执行文件中 (.exe)、动态链接库中 (.dll),也可以封装在 COM+ 组件中。

对代码或程序的封装,其目的在于实现易开发、可维护、能复用的软件模式。因为现代软件工业已经不再是过去那种依靠一两个软件英雄通过手工作坊式的编程来完成订单的模式。对大型软件项目而言,只有进行科学完善的系统分析和设计,将功能不断细分,并通过代码的封装技术,提高封装对象的可复用性、可维护性,才能适应日益复杂的业务上和性能上的需求。

在目前主流的应用程序设计中,我们可以发现具有界面和业务分层设计的系统往往更具有维护和扩展方面的优势。从早期的客户机/服务器系统,到流行的多层分布式系统,其核心思想都是强调了界面和业务的分离。几乎所有的程序员都知道程序界面和业务逻辑分离的好处,但在如何实现程序界面和业务逻辑分离的问题上,不少程序员却很为难。甚至有人还会有这样的疑问:

- 如何实现分离,分离模块之间如何通信?
- 实现分离会不会让我放弃 RAD,比如:我无法方便地使用数据库控件?
- 实现分离会不会很难,需要写很多代码吗?

第一个问题显然是从传统的面向过程思维来看待分离,其实以前程序员也努力将程序分解成模块,并通过函数来彼此调用,但彼此通信一直是个头疼的问题。特别是设计良好的分离模块需要保持内部紧聚合,外部松耦合,实现不太容易。但在 OOP 中,我们的眼中只有对象,对象之间存在灵活有效的通信机制,对象在逻辑上就是离散的、封装良好的功能模块,所以无论是实现分离还是彼此通信都很容易。

第二个问题比较有代表性。大多数速成的程序员都是从拖放控件起家的。RAD (快速应用开发) 本身并没有错,问题是编程不能只依赖于 RAD。Delphi 专家 Marco Cantu 说过 “Choosing object-oriented programming means leaving some visual programming practices behind you”,这意味着我们需要找到 RAD 和 OOP 之间的平衡点,以面向对象的思维方式来利用 RAD。因为 Delphi 不是 C++, 所以我们不会放弃 RAD。后面大家会看到这样的例子。

第三个问题我觉得是个信心的问题。如果我们注意学习新技术、新方法,实际上很多看似

很难的东西并不可怕。Delphi 为我们提供了很多帮助，使得我们不用写很多代码，但其中仍然有很多窍门需要去学。

后面我通过具体的示例程序来讲解如何实现程序界面和业务逻辑的分离，从这个例子读者可以看到界面和业务分离的演化过程。

8.1.2 从界面和业务分离到分布式多层体系结构

一个设计良好的应用程序在逻辑上应该至少划分为界面和业务两个层次。界面和业务的分离将十分有利于系统的维护和扩展，体现灵活、复用的风格。一个面向对象的程序比传统的面向函数和过程的程序更易于实现界面和业务的分离。

首先，对于 GUI 界面，不少优秀的开发工具都以面向对象的类库封装了大量复杂的 API。我们可以在 VCL 这样庞大精致的类库支持下通过 GUI 组件来构造界面。这样，通过界面对象的使用，不但可以满足用户对界面的苛刻要求，还可以让界面对象专注于用户交互和操作导航，适应多变的业务需求，而不是时时受到业务变化的影响。

其次，将业务逻辑从界面中剥离出来，可以进一步细化和分解业务。通过将业务分解成不同的业务对象，可以提高对象间的内聚力，降低耦合度。这样一来，分别针对不同业务对象进行的维护和管理，可以避免牵一发而动全身的风险。

最后，从不断发展的眼光来看，分离的界面和业务可以互不干扰地升迁到新的结构体系中，适应新技术潮流，最终实现分布式跨平台的应用。

可以说，界面和业务的分离，是一个可扩展可重用系统设计的基础，也是面向对象思想的集中体现。

界面和业务的分离经由逻辑分离、物理分离直至空间分离，最终实现了分布式多层体系结构，成为企业级应用的主流趋势。

通常分布式系统划分为 3 层，即：表现层（或称为表示层）、业务层和数据层，如图 8-1 所示。表现层用于和用户交互，它提供用户界面及操作导航服务。业务层用于业务处理，提供商业逻辑等各种约束。数据层用于数据的集成存储，这些数据即可以是平面文件，也可以是 RDBMS^①管理的数据。

表现层有两种主要的用户界面选项：Win32 客户机和基于浏览器的客户机。

Win32 客户机是最容易创建的，且其提供了一个更丰富的用户界面。它通常由可视化的开发工具（如 Delphi）创建。不足之处是，这种客户端软件很难安装和维护，它需要在每台客户机上安装；需要升级时，必须对每台客户机进行更新。除了为客户机安装软件以及维护/更新时很麻烦外，还有另一个很难解决的问题：由于安装在客户机上的操作系统和其他软件版本不同，客户机上的 DLL 冲突很频繁。这些冲突很难诊断和解决，被称为“DLL Hell”。

基于浏览器的客户机较难创建，且其仅提供了一个比较有限的用户界面，只包含几个控件，并且对屏幕布局和屏幕事件处理的控制也很少。但它们很容易安装，所有客户机只需要一

① RDBMS 是关系型数据库管理系统 (Relational Database Management System)，是用来存储和管理数据库的引擎。著名的 RDBMS 有 Oracle、SQL Server 等。

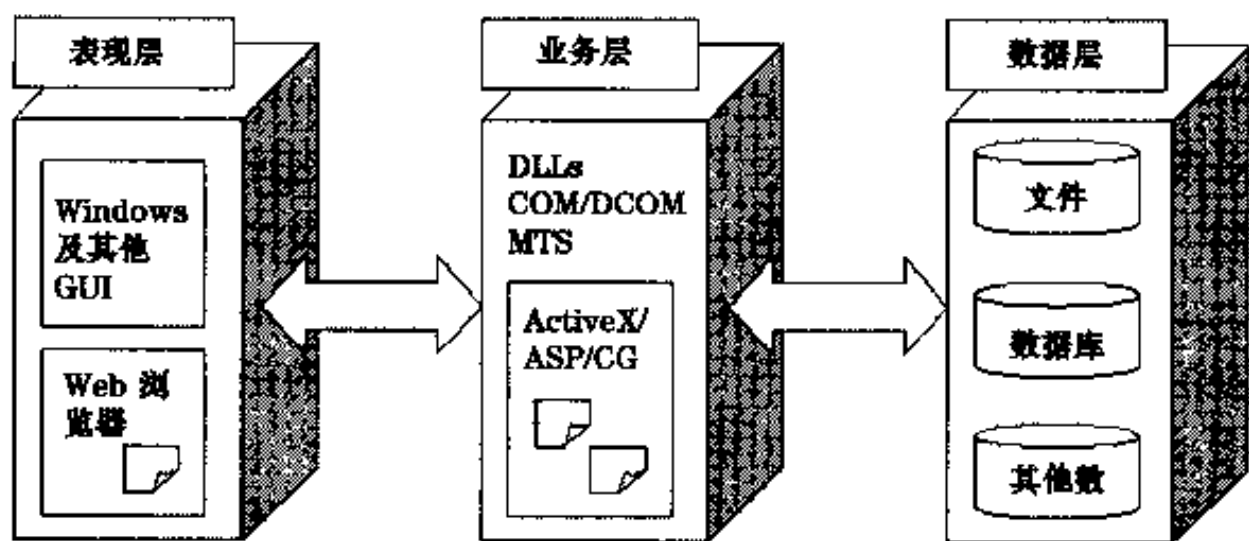


图 8-1 体系结构通常划分为 3 层

个兼容的浏览器和一个 Internet 或 Intranet 连接。

基于浏览器的客户机可以使用一些技术（如客户端脚本或者 Java 小程序，以及最新的 Web Form 技术），以使用户界面更丰富，功能更强。这些方法都适用于当前大多数浏览器。

一般情况下，基于网页的解决方案非常适用于基于 Web 浏览器的 Intranet/Internet 应用程序。它易于维护、使用方便、跨系统跨平台等诸多优势已经越来越受到人们的追捧，成为一种很有竞争力的解决方案。

业务层也叫事务逻辑层或中间层，是应用程序的脉搏。它负责在分布式系统的中间层处理数据，那里有事务处理规则和业务流程约束数据的处理。这一层主要用于大批量处理、事务支持、大型配置、信息传送和网络通信。由此可以看出这一中间层很复杂。

在分布式系统中，我们可以把复杂业务关系细分为多项功能单一的服务，每项服务都执行一项特殊任务。这些服务可以用相对独立的服务组件来实现其功能。通过分布这些组件，我们可以平衡数据处理负载，协调业务逻辑关系，调整业务规模和业务规则。

这一切在技术上都是可以实现的，例如：COM[⊖]负责基本的 COM 和事务处理功能。IIS 负责 Internet 服务，如 SMTP（Simple Mail Transfer Protocol）、FTP（File Transfer Protocol）和 HTTP（Hyper-Text Transfer Protocol）。每项服务都可以享用其他服务和创建新服务。

业务层是很重要的。它包含了目前提供特殊服务的数目最大的组件对象。这种灵活性是大型的企业应用程序所需要的，它可以根据表现层的用户请求，从数据层获取、处理并返回数据，以响应用户需求。

许多开发者更喜欢把业务层分为三个子层，从而创建了五层体系结构，其中三个子层为：

- 外观服务层，或应用程序服务层、工作流层。负责与表现层进行通信，实现应用程序用户界面外观的服务。
- 主业务服务层，包含各种业务对象、服务对象，完成业务逻辑。
- 数据库服务层，这一层专门负责与数据库通信，负责建立 SQL 语句和调用存储过程，

⊖ COM⁺——COM⁺是 COM 的扩充。通过添加服务，扩展了 COM，使其可以用于企业中。而 COM 只提供了组件框架。要使 COM 组件有事务处理能力，还需要附加的服务。这是 COM⁺的目的。COM⁺现已在微软的 Windows 2000/XP 中使用。其中 Windows 2000 中使用的是 COM⁺ 1.0 版，Windows XP 使用的是 COM⁺ 1.5 版。

实现对数据库的访问。

把业务层分成外观服务层、主业务服务层和数据库服务层的好处是：外观层更接近用户应用程序而不是业务逻辑，而主业务仅包含了实际的业务逻辑。因此，能够根据特定用户的应用程序来设计外观层，从而在应用程序使用中，按照需求的变化灵活改变外观层的类，以便实现新的界面。比如，用户需要将传统的 Windows 窗体界面更换到 Web 网页界面时，我们可能需要修改的仅仅是外观层中对应类的接口而已。因为主业务类实现的是抽象的业务逻辑，而不是任何特定的应用程序功能，所以它能设计出更好的重用性。由于它不像外观层的类那样允许自由修改，因此保持了健壮性。同样，数据库服务层把主业务从数据库的复杂逻辑中分离出来，便于有效地使用 COM+ 或 MIDAS 来管理事务，从而进一步将组件中事务部分从非事务部分分离出来。同时，数据库服务层专门负责与数据库通信，负责建立 SQL 语句和调用存储过程，实现对数据库的访问。Delphi 中的远程数据模块实际上就起到了数据库服务层的功能。

数据层实际上就是资源管理层。与业务层相比，没有或较少有数据的处理。而是定义了大量数据的管理任务。数据库和资源会变得越来越来多，因此，这项任务也变得越来越困难。

通常数据层使用大型的 RDBMS 来管理，如：Oracle。使用 RDBMS 来管理数据的好处是可以协助数据的处理，提高数据的使用效率。

Ian Graham 认为“现有的结构化方法对于分布式系统不仅几乎没有什么贡献，反而在实际中阻碍了它的发展。而对象技术提供了模型化分布式系统的一种自然方法。”（引自《面向对象方法：原理与实践》，机械工业出版社 2003 年 3 月引进出版。）

在分布式系统中，由于网络和中间件技术解决了通信上的问题，因此网络节点可以看成是对象或对象集，业务对象在网络上的分布是透明的，客户端用户请求业务对象服务时不需要关心该对象的所在位置，网络间的通信变成了对象间的通信。这就是 Ian Graham 认为的对象模型所提供的“自然方法”。

显然，“人们越来越重视分布式系统，由于对象技术具有封装和消息传递的特点，因此对象技术可能是最适合采用的方法。”（引自 Ian Graham 的《面向对象方法：原理与实践》一书。）

然而从界面和业务分离到分布式多层体系结构并不是一蹴而就的，关于这方面的讨论有太多的理论文章，但对于程序员来说如何实现才是最关键的。

在本章中，我专门设计了一个界面和业务分离的演化实例，通过开发实践来帮助读者体会界面和业务分离的重要性，并掌握 Delphi 的常用实现方法。最后通过业务跨平台和界面跨平台的解决方案和开发实例，展示了分布式应用中的最新潮流，以及 Delphi 支持新技术方面的强大功能。

8.2 界面和业务分离的演化实例

8.2.1 一个典型的 RAD 程序

在许多数据库应用程序中，我们都需要有一个数据记录的维护模块。图 8-2 就是一个典型的用户维护的数据库应用程序。该程序提供了一个模糊查询功能，系统管理员可以查询到一个符合条件的用户集，并能选择其中某个用户，对其详细信息进行维护。

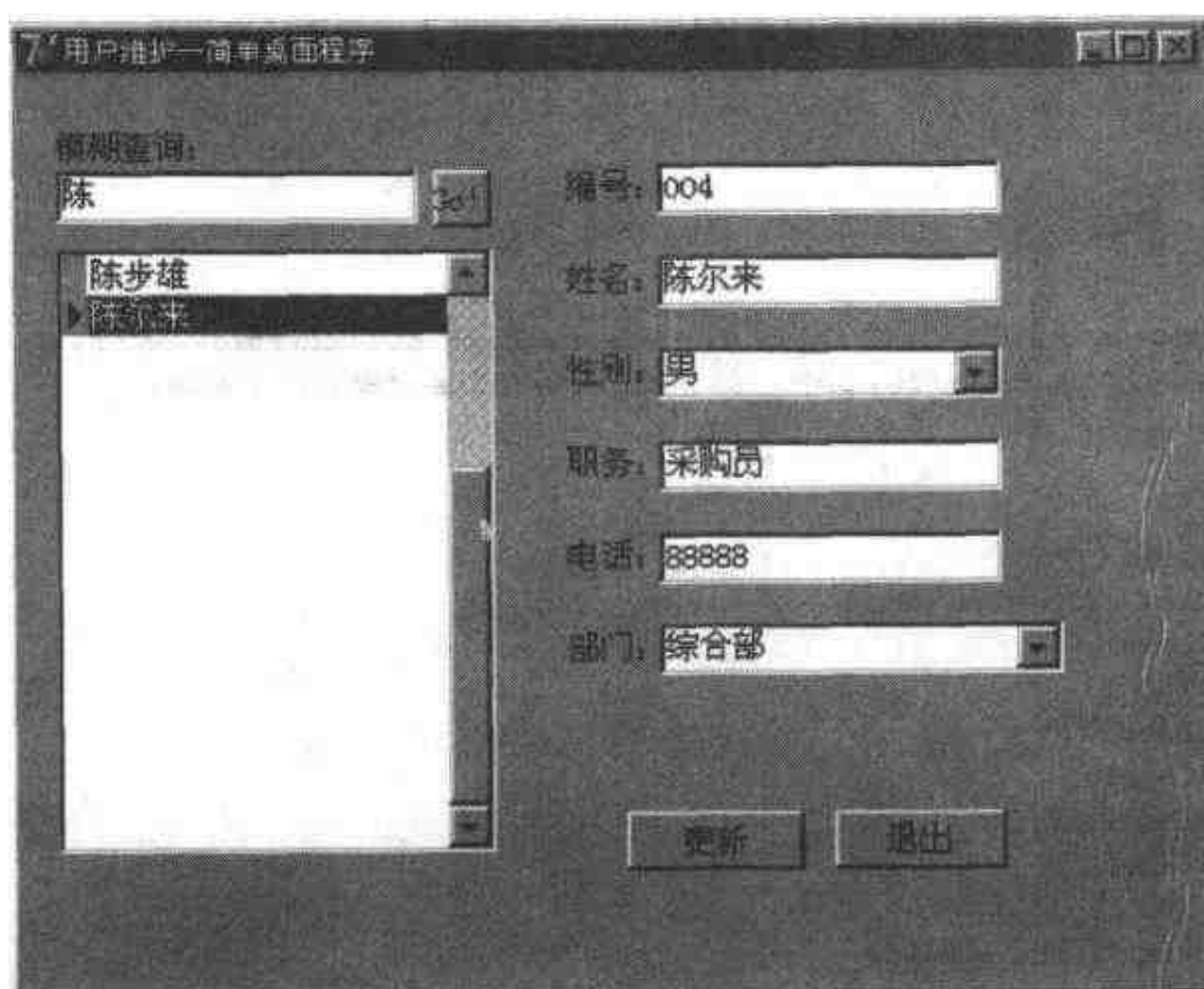


图 8-2 用户维护——一个数据记录的维护程序

显然，很多 Delphi 程序员都有设计这样一个数据库程序的经验。按照 RAD 的方法，我们可以在 Form 上拖放一些数据库控件，如图 8-3 所示。并在一些响应事件中填写代码，最后完成程序，如示例程序 8-1 所示。对于熟练的程序员，整个过程用不了半小时。

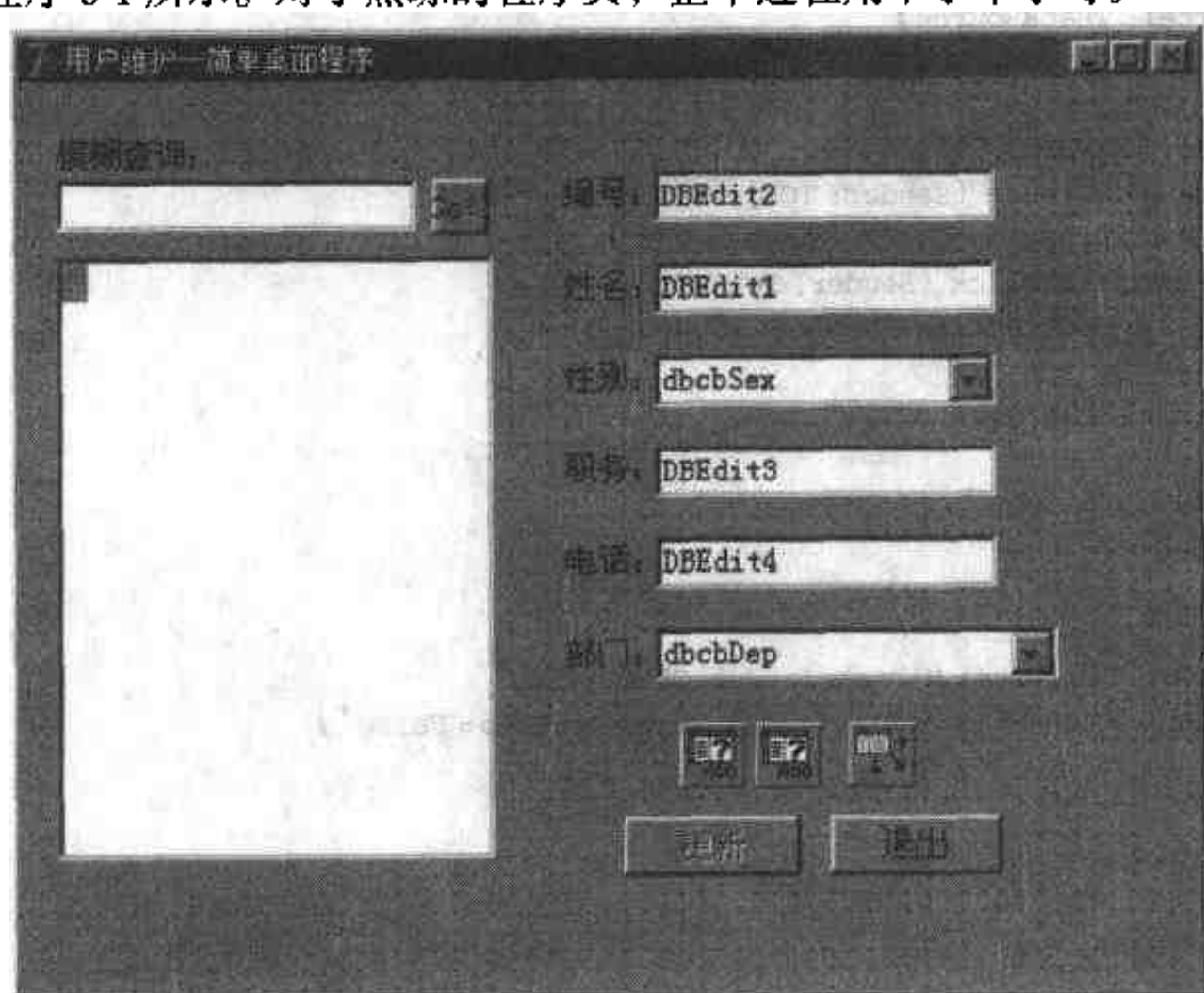


图 8-3 用户维护——简单桌面程序的 RAD 设计界面

 示例程序 8-1 用户维护——一个简单桌面程序的源代码

```

unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, ADODB, ExtCtrls, DBCtrls, StdCtrls, Grids, DBGrids, Mask,
  Buttons;

type
  TForm1 = class (TForm)
    btnQryByName: TSpeedButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    btnExit: TButton;
    edtQryByName: TLabelledEdit;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
    DBEdit4: TDBEdit;
    DBGrid1: TDBGrid;
    dbcbSex: TDBComboBox;
    dbcbDep: TDBComboBox;
    DataSource1: TDataSource;
    adqDep: TADOQuery;
    adqUser: TADOQuery;
    btnUpdate: TBitBtn;
    procedure FormCreate (Sender: TObject);
    procedure btnQryByNameClick (Sender: TObject);
    procedure btnExitClick (Sender: TObject);
    procedure btnUpdateClick (Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

const
  ADO_STRING = ' Provider = Microsoft.Jet.OLEDB.4.0;'
    + ' Data Source = wz.mdb; Persist Security Info = False';

var
  Form1: TForm1;

implementation

{$R *.dfm}
  
```

```
procedure TForm1.FormCreate (Sender: TObject);
var
  i: Integer;
  tmpstrs: TStringList;
begin
  tmpstrs := TStringList.Create;
  with adgDep do
  try
    ConnectionString := ADO_STRING;
    sql.Clear;
    sql.add ('select * from M_BMBM');
    Open;
    for i := 1 to RecordCount do
    begin
      tmpstrs.Add (Fieldbyname ('BMMC') .AsString);
      Next;
    end;
    Close;
  finally
    Close;
  end;
  dbcbDep.Items := tmpstrs;
end;

procedure TForm1.btnQryByNameClick (Sender: TObject);
begin
  with adgUser do
  begin
    close;
    ConnectionString := ADO_STRING;
    Parameters.ParamByName ('name') .value := '%' + edtQryByName.Text + '%';
    open;
  end;
  btnUpdate.Enabled := true;
end;

procedure TForm1.btnExitClick (Sender: TObject);
begin
  close;
end;

procedure TForm1.btnUpdateClick (Sender: TObject);
begin
  adgUser.Post;
end;

end.
```

这样的代码对于初学编程的新手做练习用还问题不大，但是在很多专业的应用程序开发中，出现这样的代码就很危险了。因为这里将用户界面和业务逻辑混在一起。如果出现问题需要维护，业务代码的修改会同时牵动到界面代码。显然，这是没有经过设计的（至少没有体现设计思想）的随意写法。虽然该程序可以实现既定的功能，但这是写死的程序，没有灵活性。

这种应用程序是一次性程序，没有一点可重用的价值。这里的例子还比较简单，但如果界面和业务都很复杂，这样的写法是致命的。我就见过一位程序员在一个按钮的 click 事件中写了上百行的业务逻辑代码。这是何等的恐怖，这样的程序无论是阅读还是修改都是难以想像的。

8.2.2 界面和业务的逻辑分离

为了将示例程序 8-1 设计成界面和业务的逻辑分离的应用程序，我们首先要将程序中的逻辑划分成外观类和业务类（对于数据库应用程序而言，最好再划分一个数据库访问类），如图 8-4 所示。外观类通常是 GUI 界面对象，如 Windows 窗体。它可以通过 RAD 的方式可视化地完成设计。在这个程序中，我使用了 TfrmUsers 作为外观类，其设计如图 8-5 所示。注意这里只提供交互界面，不提供业务实现。

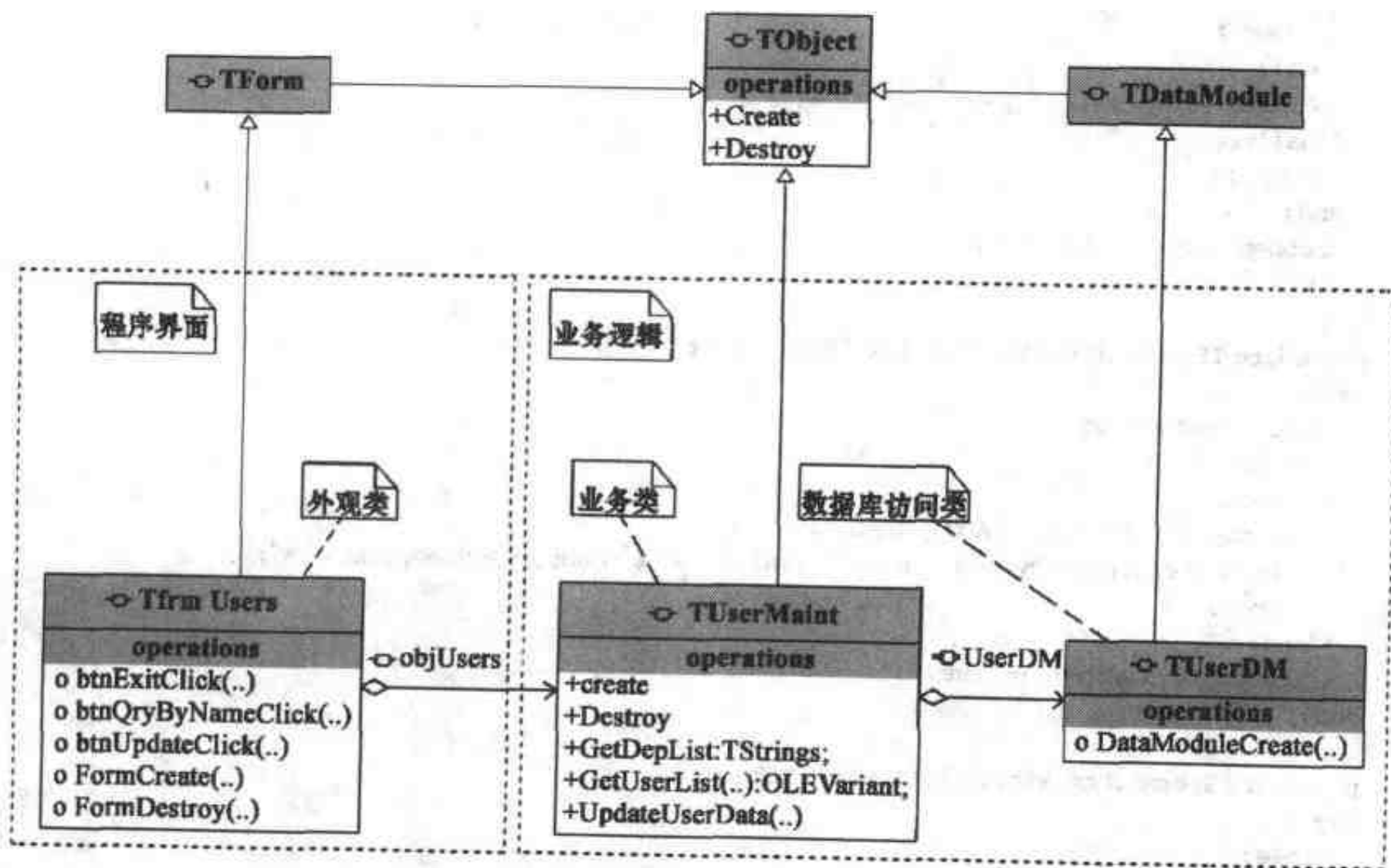


图 8-4 界面和业务的逻辑分离的设计

业务类用于实现业务逻辑，那里有事务处理规则和业务流程约束数据的计算。在这个程序中，我使用了 TUserMaint 作为业务类，并把这个类放在一个普通的 Pascal 单元文件中，TUserMaint 是我自己设计的非可视化类，无法像控件一样进行拖放。

对于数据库应用程序，数据库访问类可以集中完成数据集和数据库的连接。这里我使用了 TUserDM 类作为数据库访问类，它是一个数据模块（TDataModule 的派生类）。TUserDM 类是一个容器，我们可以使用 RAD 的方法可视化地在其中添加和设置数据库存取组件，如图 8-6 所示。读者可以看出，这里我使用的是 ADO 数据库存取组件。由于使用了数据库访问类，一旦我们需要更换数据库（如换成 Oracle 或 SQL Server）或数据库存取技术（如换成 DBE 或 DBExpress

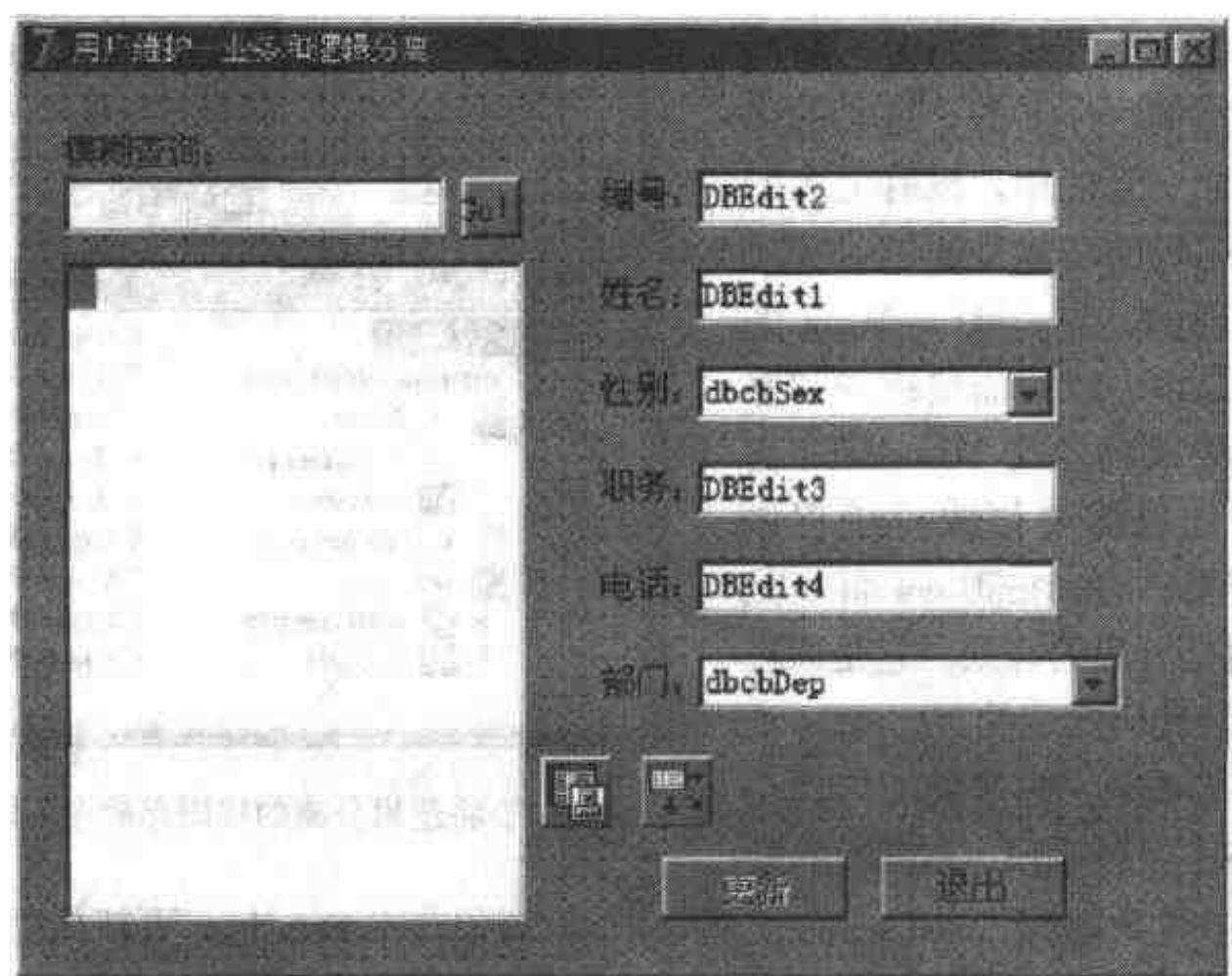


图 8-5 通过 RAD 的方式可视化地完成外观类 TfrmUsers 设计

存取技术) 只要在这里做修改就行, 而不会影响整个程序。因为我们没有直接访问数据集组件 (如: TADOQuery), 而是间接地通过 TDataSetProvider 组件来访问数据集的, 至于 TDataSetProvider 组件连接的是 TADOQuery 还是 TTable, 对业务和界面部分都没有关系。

这里实际上涉及到一个灵活使用 RAD 的问题, 虽然我们是采用完全面向对象的思想来编程, 但通过拖放组件提高了编程的效率。不少自称高手的 Delphi 程序员武断地认为只要使用数据库组件就无法实现面向对象的思维, 无法将业务和界面分离, 从而反对使用数据库组件。

这个例子就是专门针对数据库设计的。我们已经看到了在数据库访问类 TUserDM 中, 我使用了数据库访问组件和数据集组件; 其实, 细心的读者在外观类 TfrmUsers 上也不难发现我使用了数据感知控件, 最重要的是我还使用了 TClientDataSet 组件。由于 Delphi 提供了 DataSnap 技术, 使得 TClientDataSet 组件和 TDataSetProvider 组件互相结合, 完成数据封包、传递和解析的过程, 因此可以将数据集和数据显示及处理分离开, 以便实现数据库应用上的逻辑分离。由于使用了数据库相关的组件, 使得我省去了大量的繁琐编码, 因此读者可以看到最终的程序十分简洁。

在完成了代码逻辑上的分离设计后, 我们最好将这些逻辑分离的代码存放在不同的单元文件中, 如图 8-7 所示。这样既便于修改、维护, 又可以进一步通过封装实现物理上、空间上的分离。

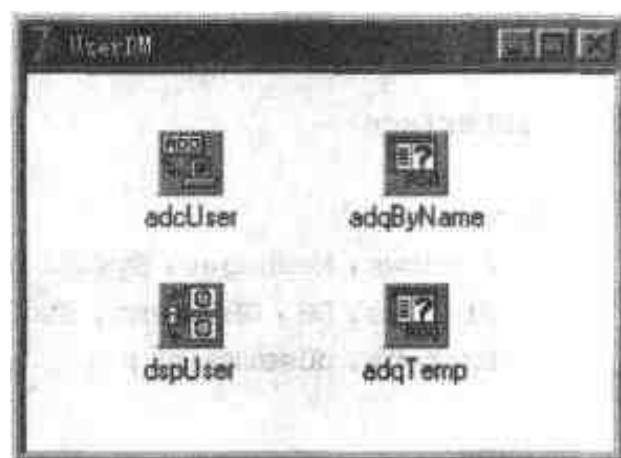


图 8-6 在数据模块中添加数据库存取组件

我们现在来看一下经过逻辑分离后的界面单元代码（示例程序 8-2），这里的代码仅供用户界面交互使用，没有更多的逻辑。其中业务逻辑的功能是通过业务对象 objUsers 来实现的，它是 TUserMaint 的一个实例。这就是说，以后需要维护和修改业务逻辑，需要改动的是 TUserMaint 而不是 TfrmUsers。如果需要修改应用程序的外观，也只需要改动 TfrmUsers 而不是 TUserMaint。这就是分离的好处，它是通过类来抽象逻辑封装代码实现的。

在示例程序 8-2 中，数据集的传递是通过 TClientDataSet 组件 cdsUserMaint 的数据封包实现的。查询数据时，cdsUserMaint 从 objUsers 对象获得 Data 包；更新数据时，cdsUserMaint 将数据更新变化装在 Delta 包中传出，完成数据库更新。

示例程序 8-2 逻辑分离后的界面单元代码

```
unit ufrmUsers;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBClient, StdCtrls, DBCtrls, Grids, DBGrids, Mask, ExtCtrls,
  Buttons, uUserMaint;

type
  TfrmUsers = class (TForm)
    btnExit: TButton;
    btnQryByName: TSpeedButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    edtQryByName: TLabeledEdit;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
    DBEdit4: TDBEdit;
    DBGrid1: TDBGrid;
    dbcbSex: TDBComboBox;
    dbcbDep: TDBComboBox;
    DataSource1: TDataSource;
    cdsUserMaint: TClientDataSet;
```

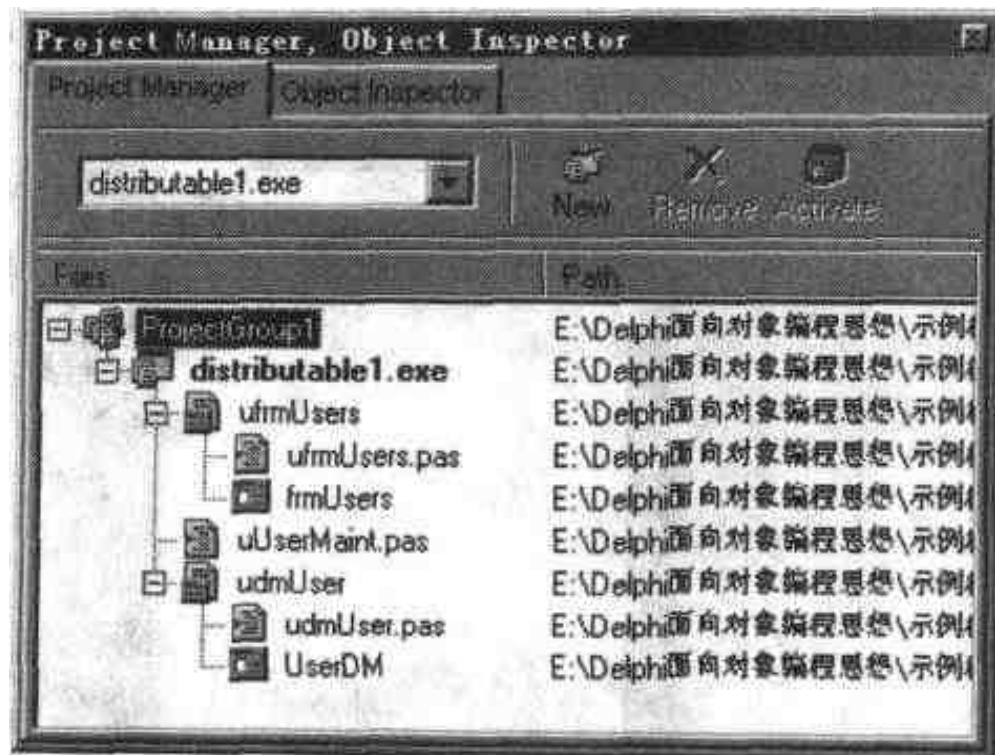


图 8-7 将逻辑分离的代码存放在不同的单元文件中

```

cdsUserMaintID: TWideStringField;
cdsUserMaintNAME: TWideStringField;
cdsUserMaintSEX: TWideStringField;
cdsUserMaintJOB: TWideStringField;
cdsUserMaintTEL: TWideStringField;
cdsUserMaintCALL: TWideStringField;
cdsUserMaintDEP: TWideStringField;
cdsUserMaintGROUP_ID: TWideStringField;
cdsUserMaintPASSWORD: TWideStringField;
btnUpdate: TBitBtn;
procedure btnUpdateClick (Sender: TObject);
procedure btnQryByNameClick (Sender: TObject);
procedure FormCreate (Sender: TObject);
procedure btnExitClick (Sender: TObject);
procedure FormDestroy (Sender: TObject);
private
    objUsers: TUserMaint;
public
    {Public declarations}
end;

const
    M_TITLE = '操作提示'; //所有提示对话框的标题

implementation

{$R *.dfm}

procedure TfrmUsers.btnUpdateClick (Sender: TObject);
var nErr: integer;
begin
    if cdsUserMaint.State = dsEdit then cdsUserMaint.Post;
    if (cdsUserMaint.ChangeCount > 0) then
    begin
        objUsers.UpdateUserData (cdsUserMaint.Delta, nErr);
        if nErr > 0 then
            application.MessageBox ('更新失败!', M_TITLE, MB_ICONWARNING)
        else
        begin
            application.MessageBox ('更新成功!', M_TITLE, MB_ICONINFORMATION);
            btnQryByNameClick (nil);
        end;
    end;
end;

procedure TfrmUsers.btnQryByNameClick (Sender: TObject);
begin
    btnUpdate.Enabled := true;
    dbcbDep.Items.AddStrings (objUsers.GetDepList);
    cdsUserMaint.Active := false;
    cdsUserMaint.Data := objUsers.GetUserList (edtQryByName.Text);
    cdsUserMaint.Active := True;
end;

```

```

procedure TfrmUsers.FormCreate (Sender: TObject);
begin
    objUsers: = TUserMaint.Create;
end;

procedure TfrmUsers.btnExitClick (Sender: TObject);
begin
    close;
end;

procedure TfrmUsers.FormDestroy (Sender: TObject);
begin
    objUsers.Free;
end;

end.

```

示例程序 8-3 是逻辑分离后的业务单元代码。这里完成查询和维护数据记录的业务。需要说明的是我这里仅仅是一个简单的示例程序，没有包括数据的合法性检查、事务处理、异常处理等许多重要逻辑。在实际系统中，业务会更复杂。

示例程序 8-3 逻辑分离后的业务单元代码

```

unit uUserMaint;
interface

uses
    Windows, Messages, SysUtils, Variants, Classes, DBClient, udmUser;

type
    TUserMaint = class(TObject)
    private
        UserDM: TUserDM;
    public
        function GetDepList: TStrings;
        function GetUserList (strName: String): OLEVariant;
        procedure UpdateUserData (UserData: OleVariant; out ErrCount: Integer);
        constructor create;
        destructor Destroy; override;
    end;

implementation

|
| ***** TUserMaint
|
constructor TUserMaint.create;
begin
    UserDM: = TUserDM.Create (nil);
end;

destructor TUserMaint.Destroy;
begin
    freeandnil (UserDM);
end;

```

```
    inherited;
end;

function TUserMaint.GetDepList: TStrings;
var
    i: Integer;
    tmpstrs: TStrings;
begin
    tmpstrs := TStringlist.Create;

    with UserDM do
    try
        if not adcUser.Connected then
            adcUser.Connected := True;
        adqTemp.sql.Clear;
        adqTemp.sql.add ('select * from M_BMBM');
        adqTemp.Open;
        for i := 1 to adqTemp.RecordCount do
            begin
                tmpstrs.Add (adqTemp.Fieldbyname ('BMMC') .AsString);
                adqTemp.Next;
            end;
        adqTemp.Close;
        result := tmpstrs;
    finally
        adcUser.Connected := False;
    end;
end;

function TUserMaint.GetUserList (strName: String): OLEVariant;
begin
    with UserDM do
    try
        if not adcUser.Connected then
            adcUser.Connected := True;
        with adqByName do
            begin
                close;
                Parameters.ParamByName ('name') .value := '%' + strName + '%';
                open;
                result := dspUser.Data;
            end;
        finally
            adcUser.Connected := False;
        end;
    end;
end;

procedure TUserMaint.UpdateUserData (UserData: OleVariant;
    out ErrCount: Integer);
begin
    UserDM.dspUser.ApplyUpdates (UserData, 0, ErrCount);
end;

end.
```

由于使用了数据库组件，在数据模块单元 udmUser（示例程序 8-4）中几乎没有什么额外的代码。惟一的代码是为了设置 ADO 连接字符串：

```
adcUser.ConnectionString: = ADO_STRING;
```

虽然高版本的 Windows 系统都带有 ADO 驱动程序，用户可以不必要像 BDE 或 ODBC 那样进行数据库设置，但数据源的位置还需要指定。ADO 连接字符串 ADO_STRING 中 “Data Source = wz.mdb” 表示程序使用的 Access 数据库 wz.mdb 保存在和应用程序相同的目录下。我没有在 TADOConnection 组件中直接设置 ConnectionString，而是将其放在一个字符串常量 ADO_STRING 中，这是为了方便读者修改。因为我不知道读者会把光盘中的示例程序复制到哪里。

示例程序 8-4 数据模块单元 udmUser

```
unit udmUser;

interface

uses
  SysUtils, Classes, DBClient, Provider, DB, ADODB;

type
  TUserDM = class (TDataModule)
    adcUser: TADOConnection;
    adqByName: TADOQuery;
    dspUser: TDataSetProvider;
    adqTemp: TADOQuery;
    procedure DataModuleCreate (Sender: TObject);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

const
  ADO_STRING = ' Provider = Microsoft.Jet.OLEDB.4.0; Data '
    + ' Source = wz.mdb; Persist Security Info = False';

implementation

{$R *.dfm}

procedure TUserDM.DataModuleCreate (Sender: TObject);
begin
  adcUser.ConnectionString: = ADO_STRING;
end;

end.
```

现在如果我们运行经过修改后的这个程序，发现和原先程序的功能没有什么两样。有的读者可能会抱怨，既然如此，何必要把事情搞得那么复杂，多出了很多代码和单元文件。

是的，如果你打算写一个桌面小程序，当然不用进行这样复杂的考虑。如果你是要完成一

个重要的商业软件产品或是一个大型的项目工程，就不得不养成一个这样考虑问题的习惯——必须在设计时充分考虑逻辑上的分离与独立，每一个环节避免界面和业务代码纠缠在一起的实现。这如同建一个狗窝和建一幢高楼的区别。对于前者你可以快速搭建，不用设计，大不了推翻重建；但对于建高楼你却要小心谨慎，因为你付不起重建的代价。

另外，将分离出来的独立的逻辑划分到不同的单元文件中也是一个好习惯。除了修改、维护上的便利外，还可以进一步进行物理上的封装，以轻松实现 C/S（客户/服务器）结构或多层结构。

8.2.3 界面和业务的物理分离

我用 ModelMaker 来进行界面和业务的物理分离的设计，如图 8-8 所示。我将界面部分设计成一个瘦客户机的形式，这是一个供用户交互的可执行文件（distributable2.exe），它封装了外观类 TfrmUsers。我把业务部分（包括数据模块）设计成提供服务的服务器，这是一个动态链接库文件（UserSvr.dll），它封装了业务类 TUserMaint 和数据库访问类 TUserDM。

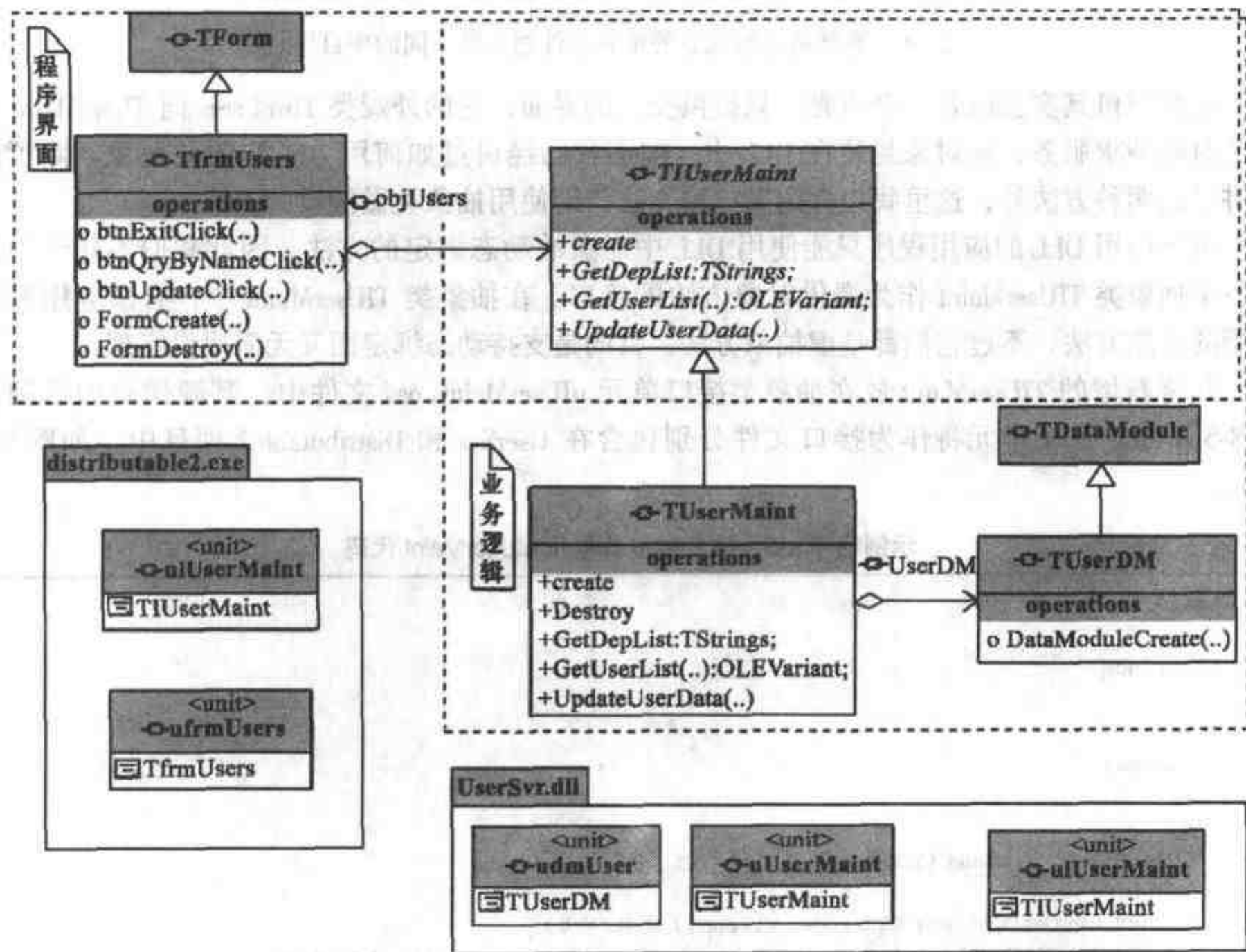


图 8-8 界面和业务的物理分离的设计

由于原来的逻辑独立的类和代码存放在不同的单元文件中，因此很容易重新将它们划分到不同的项目里，如图 8-9 所示。

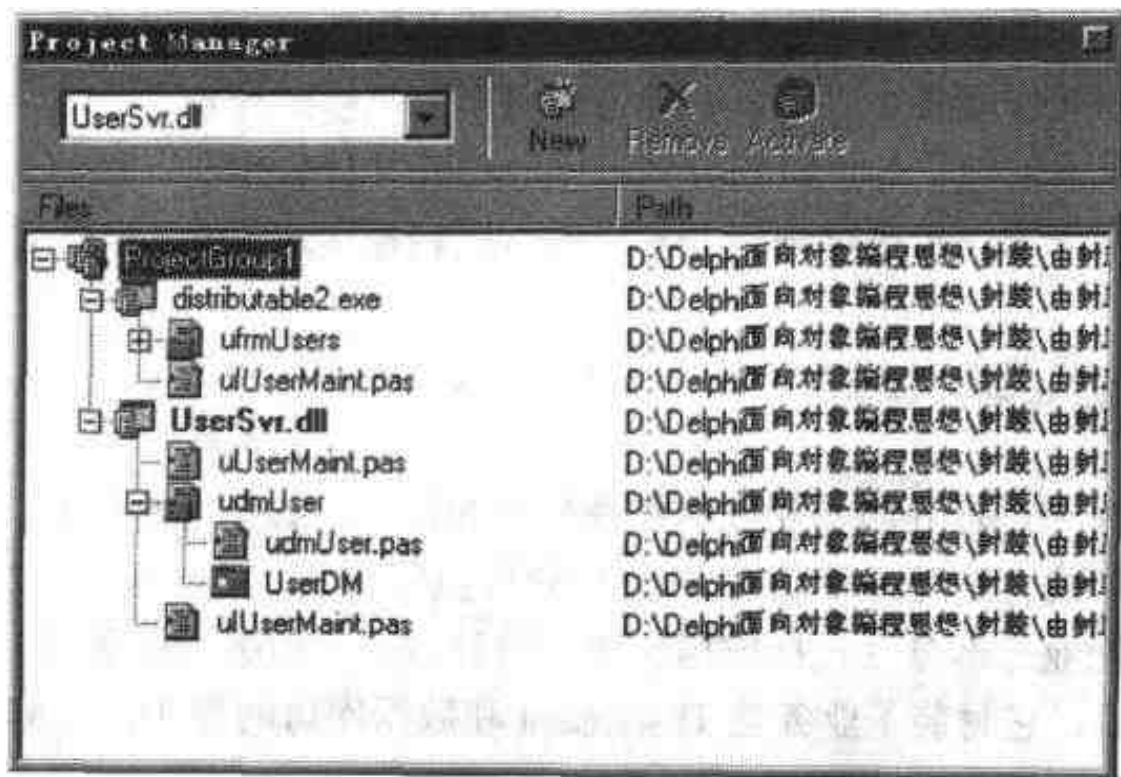


图 8-9 重新将逻辑独立的单元文件划分到不同的项目里

瘦客户机其实上就是一个空壳，只提供交互的界面，它的外观类 `TfrmUsers` 向 `TUserMaint` 的实例对象请求服务，该对象封装在 DLL 中。前面我已经讲过如何用 DLL 来封装对象。除了前面讲过的两种方法外，这里我想介绍第三种方法，即使用抽象类做接口的方法。

由于调用 DLL 的应用程序只能使用 DLL 中对象的动态绑定的方法，因此我们不妨专门设计一个抽象类 `TIUserMaint` 作为提供对象方法的接口。在抽象类 `TIUserMaint` 中，有供应用程序使用的对象方法，不过它们都是虚抽象方法，目的是支持动态绑定而又无需提供实现。

我将新增的 `TIUserMaint` 放在抽象类接口单元 `uUserMaint.pas` 文件中，其源代码如示例程序 8-5 所示。这个单元将作为接口文件分别包含在 `UserSvr` 和 `Distributable2` 项目中，如图 8-9 所示。

示例程序 8-5 抽象类接口单元 `uUserMaint` 代码

```
unit uUserMaint;

interface

uses
  Classes;

type
  TIUserMaint = class(TObject)
  public
    function GetDepList: TStrings; virtual; abstract;
    function GetUserList(strName: String): OLEVariant; virtual; abstract;
    procedure UpdateUserData(UserData: OleVariant; out ErrCount: Integer);
      virtual; abstract;
    constructor create; virtual; abstract;
  end;
```

```

TIUserMaintClass = class of TIUserMaint;

implementation
//没有实现代码
end.

```

在示例程序 8-5 中还定义了 TIUserMaintClass 类型，它是 TIUserMaint 的类引用。这对于把实现类从 DLL 传递到进行调用的应用程序是必要的。

一般抽象类只定义接口，它由虚抽象方法组成而没有实际的数据。为了实现抽象类 TIUserMaint 的抽象方法，原来的 TUserMaint 类需要继承 TIUserMaint 类，并覆盖其所有的虚抽象方法。新的 TUserMaint 类声明如下：

```

TUserMaint = class (TIUserMaint)
private
    UserDM: TUserDM;
public
    function GetDepList: TStrings; override;
    function GetUserList (strName: String): OLEVariant; override;
    procedure UpdateUserData (UserData: OleVariant; out ErrCount: Integer);
        override;
    constructor create; override;
    destructor Destroy; override;
end;

```

但实际上 TUserMaint 类原有的实现部分并不需要改动，所以我们的工作量不大。

示例程序 8-6 是动态链接库 UserSvr.dll 的源代码。这里使用了 TObjUsers 函数，该函数返回了一个类型为 TIUserMaintClass 的类引用而不是对象引用，所以在应用程序中可以使用这样的代码来创建 DLL 封装的对象：

```
objUsers: = TObjUsers.Create;
```

但这并不意味着 TObjUsers 是一个类，记住这里 TObjUsers 是一个 DLL 输出的函数，它的返回类型是一个类引用类型。

示例程序 8-6 动态链接库 UserSvr.dll 的源代码

```

library UserSvr;

uses
    ShareMem,
    SysUtils,
    Classes,
    uUserMaint in 'uUserMaint.pas',
    udmUser in 'udmUser.pas' {UserDM: TDataModule},
    uIUserMaint in 'uIUserMaint.pas';

{$R *.res}
function TObjUsers: TIUserMaintClass;
begin
    result: = TUserMaint;
end;

```

```

exports
  TObjUsers;

begin
end.

```

细心的读者可能已经发现，既然 TObjUsers 函数的返回类型为 TUserMaintClass，TUserMaintClass 在示例程序 8-5 中声明为：

```
TUserMaintClass = class of TUserMaint;
```

那么 result: = TUserMaint 会不会是写错了呢？

没有写错！我们在应用程序中声明了 TUserMaint 类型的对象 objUsers，通过传递类引用，那条 objUsers: = TObjUsers.Create 语句实现的是 objUsers: = TUserMaint.Create 功能，这里面隐含了 TUserMaint 向 TUserMaint 转型的过程。当然，TUserMaint 作为抽象类本身也无法直接创建自己的实例，所以必须通过转型才行。另外，在 TUserMaint 的派生类中可以随意改变方法的实现，却不会影响到方法的接口。这就是说，你以后通过进一步修改 DLL 封装对象的实现方法来升级 DLL，无需重新修改和编译应用程序。因为 TUserMaint 作为抽象类提供的方法接口没有改变。

示例程序 8-7 是应用程序实现界面和业务的物理分离后的界面单元的源代码。这里要注意几点：

- 在 interface 部分要 Uses 抽象类接口单元 uUserMaint。
- objUsers 声明为 TUserMaint 类型。
- 声明 DLL 函数：function TObjUsers: TUserMaintClass; external 'UserSvr.dll';

除此之外，几乎不需要进行其他的改动。由此可见，从界面和业务的逻辑分离演化到界面和业务的物理分离实际上并不像想像的那样困难。

示例程序 8-7 物理分离后的界面单元代码

```

unit ufrmUsers;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBClient, StdCtrls, DBCtrls, Grids, DBGrids, Mask, ExtCtrls,
  Buttons, uUserMaint;

type
  TfrmUsers = class (TForm)
    btnExit: TButton;
    btnQryByName: TSpeedButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label5: TLabel;

```

```

Label6: TLabel;
Label7: TLabel;
edtQryByName: TLabelledEdit;
DBEdit1: TDBEdit;
DBEdit2: TDBEdit;
DBEdit3: TDBEdit;
DBEdit4: TDBEdit;
DBGrid1: TDBGrid;
dbcbSex: TDBComboBox;
dbcbDep: TDBComboBox;
DataSource1: TDataSource;
cdsUserMaint: TClientDataSet;
cdsUserMaintID: TWideStringField;
cdsUserMaintNAME: TWideStringField;
cdsUserMaintSEX: TWideStringField;
cdsUserMaintJOB: TWideStringField;
cdsUserMaintTEL: TWideStringField;
cdsUserMaintCALL: TWideStringField;
cdsUserMaintDEP: TWideStringField;
cdsUserMaintGROUP_ID: TWideStringField;
cdsUserMaintPASSWORD: TWideStringField;
btnUpdate: TBitBtn;
procedure btnUpdateClick (Sender: TObject);
procedure btnQryByNameClick (Sender: TObject);
procedure FormCreate (Sender: TObject);
procedure btnExitClick (Sender: TObject);
procedure FormDestroy (Sender: TObject);
private
    objUsers: TIUserMaint;
public
    {Public declarations }
end;

var
    frmUsers: TfrmUsers;

const
    M__TITLE = '操作提示'; //所有提示对话框的标题

implementation

{$R *.dfm}

function TObjUsers: TIUserMaintClass;
external 'UserSvr.dll';

procedure TfrmUsers.btnUpdateClick (Sender: TObject);
var
    nErr: integer;
begin
    if cdsUserMaint.State = dsEdit then cdsUserMaint.Post;
    if (cdsUserMaint.ChangeCount > 0) then
    begin
        objUsers.UpdateUserData (cdsUserMaint.Delta, nErr);
    end;
end;

```

```
if nErr > 0 then
    application.MessageBox ('更新失败!', M_TITLE, MB_ICONWARNING)
else
begin
    application.MessageBox ('更新成功!', M_TITLE, MB_ICONINFORMATION);
    btnQryByNameClick (nil);
end;
end;
end;

procedure TfrmUsers.btnQryByNameClick (Sender: TObject);
begin
    btnUpdate.Enabled: = true;
    dbcbDep.Items.AddStrings (objUsers.GetDepList);
    cdsUserMaint.Active: = false;
    cdsUserMaint.Data: = objUsers.GetUserList (edtQryByName.Text);
    cdsUserMaint.Active: = True;
end;

procedure TfrmUsers.FormCreate (Sender: TObject);
begin
    objUsers: = TObjUsers.Create;
end;

procedure TfrmUsers.btnExitClick (Sender: TObject);
begin
    close;
end;

procedure TfrmUsers.FormDestroy (Sender: TObject);
begin
    objUsers.Free;
end;

end.
```

8.2.4 界面和业务的空间分离

界面和业务的物理分离为界面和业务的空间分离提供了条件。但如果仅仅要在本机上实现空间分离, 比如: 将客户机程序和服务器程序分别放在硬盘的不同目录下, 我们前面的动态链接库封装对象的例子就已经足以实现这个目标。这里要谈的是地理空间上的分离, 即将界面程序和业务服务分布在网络上的不同机器上, 实现流行的分布式计算。

这里并不深入讨论分布式计算, 而只是继续通过前面的例子来谈论由封装实现分离的进一步演化。

如果我们使用 Windows 平台, 微软的 Windows DNA (分布式网络应用程序体系结构) 就是一个不错的方案。这个方案是基于微软的 COM+ 技术的; 该技术成熟可靠, Delphi 对其提供了很好的支持。前面已经介绍过如何使用 COM+ 来封装对象, COM+ 同样为对象分布到网络上不同的地方提供了支持。这就是说, 我们可以将业务逻辑以业务服务对象的形式, 将其作为

COM+ 组件放置在地理位置不同的服务器上, 从而实现真正意义上的空间分离。

图 8-10 是界面和业务空间分离设计的类图, 图 8-11 是界面和业务空间分离设计的实现图。从中我们可以看到 COM+ 是通过接口来通信的。值得一提的是, 这里我并没有使用 Delphi 的 MIDAS 技术 (比如使用 Remote Data Module 模板和 DCOMConnection 组件) 来实现分布式程序的设计, 而是使用了比较通用的 COM/COM+ 技术。

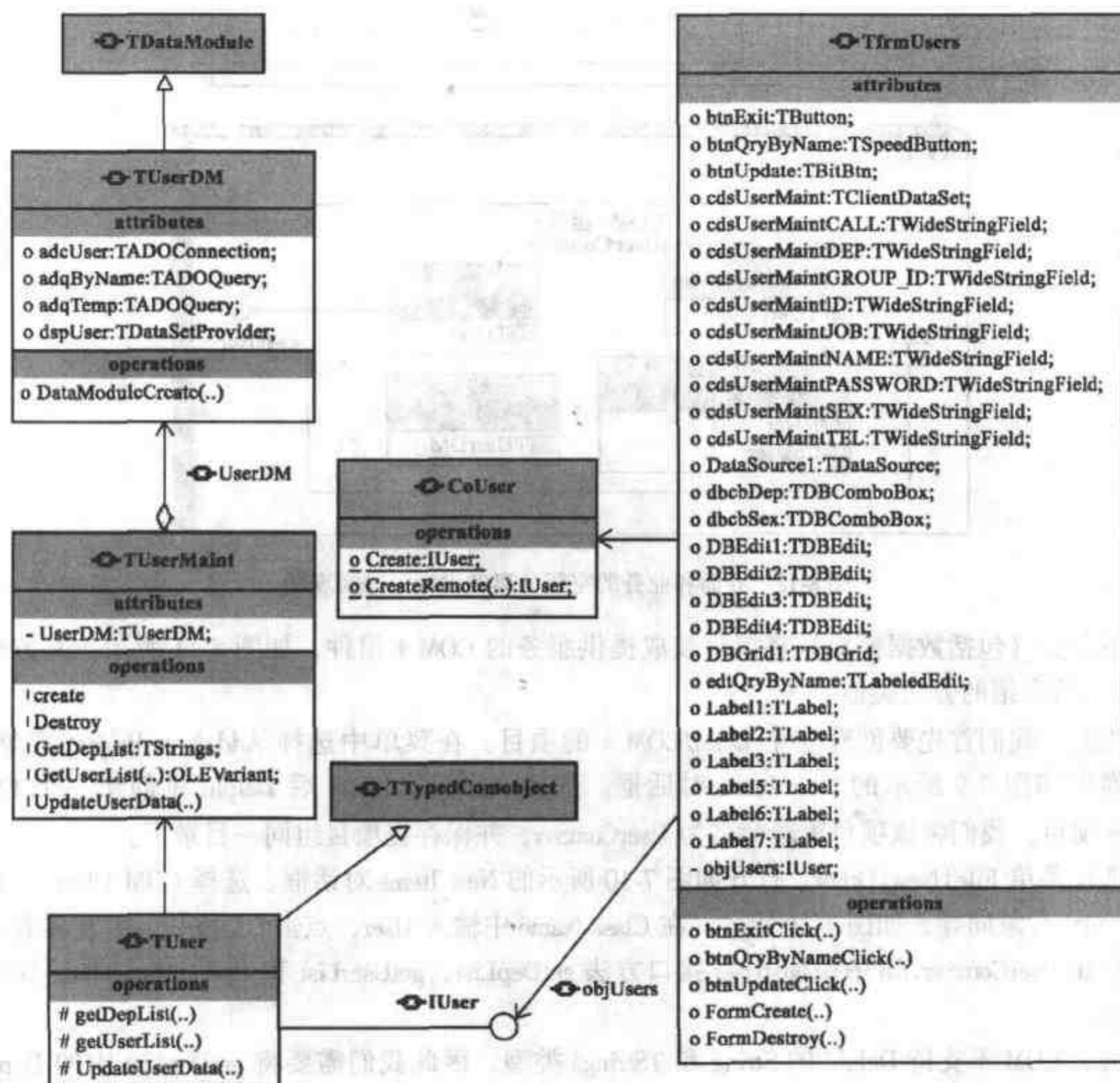


图 8-10 界面和业务的空间分离的设计——类图

国内 Delphi 程序员关于多层分布式数据库开发的知识大多来自李维的《Delphi 5.x 分布式多层应用——系统篇》。然而, 该书中的可视化 RAD 开发虽然降低了多层分布式数据库开发的门槛, 但在实际应用中却并不能满足真正面向对象开发的需要。这也是我在本例中不采用它的原因。

那么, 这个带数据库的应用程序是如何实现界面和业务空间分离设计的呢? 其实, 这里我

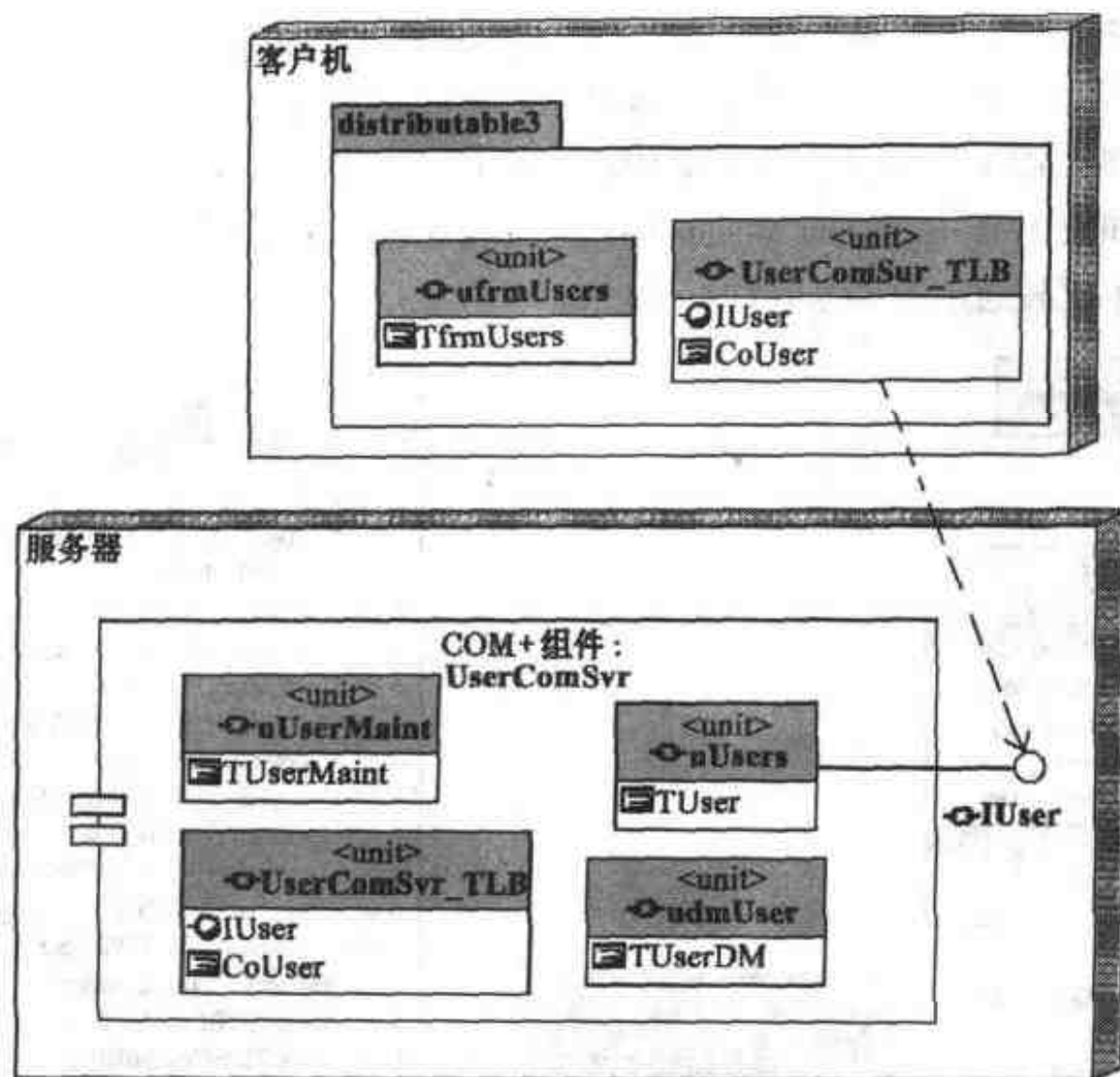


图 8-11 界面和业务的空间分离的设计——实现图

将业务部分（包括数据模块）直接封装成提供服务的 COM+ 组件，如图 8-11 所示。其方法与 7.3.3 节所介绍的方法类似。

为此，我们首先要创建一个 COM/COM+ 的项目。在菜单中选择 Add New Project 菜单项。此时弹出如图 7-9 所示的 New Items 对话框。选择 ActiveX Library 后 Delphi 将创建一个 COM/COM+ 项目。我们将该项目重新命名为 UserComSvr，并保存在项目组同一目录下。

选择菜单 File|New|Other，打开如图 7-10 所示的 New Items 对话框。选择 COM Object，此时弹出 COM 对象向导，如图 7-11 所示，在 Class Name 中输入 User，点击 OK 按钮。并在接着出现的类型库 UserComSvr.tlb 中添加 IUser 接口方法 getDepList、getUserList 和 UpdateUserData，如图 8-12 所示。

由于 COM 不支持 Delphi 的 String 和 TStrings 类型，因此我们需要将 getDepList 中的 Deps 参数改成 OleVariant 类型，将 getUserList 的 strName 参数改成 BSTR 类型。并在程序中涉及到这些参数的地方进行转换。

在示例程序 8-8 所示的由 COM 向导创建的接口单元 uUsers 中，TUser 类的作用就是实现 COM 的接口方法。其中还完成了将 GetDepList 参数由 TStrings 类型向 OleVariant 类型的转换。该转换用到了 Delphi 的可变数组。可变数组的创建和管理要调用 VarArrayXXX() 函数和过程，读者可以查看 Delphi 的联机帮助。该数组使用方便，不易出错，特别适合在 COM/COM+ 中使用。

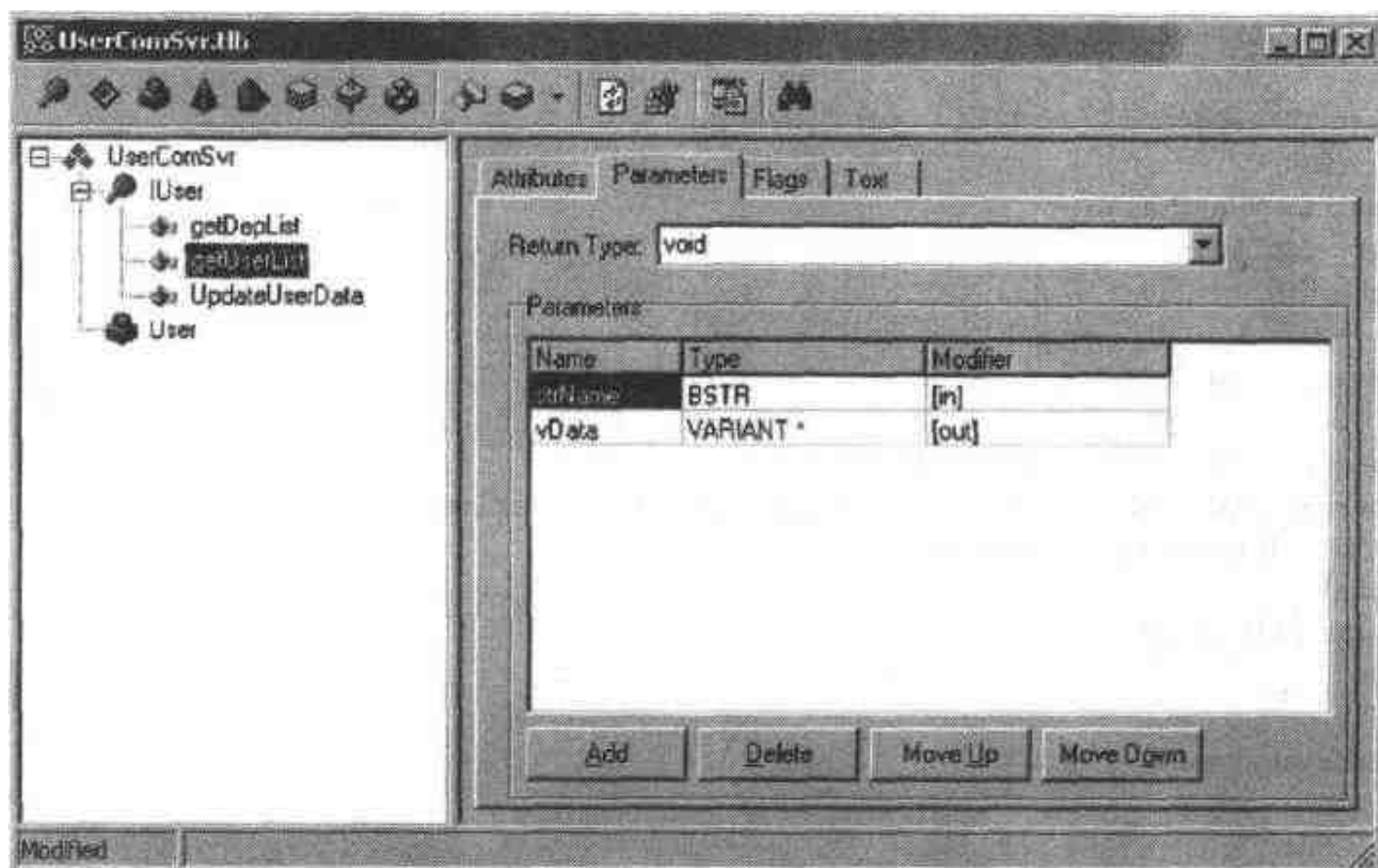


图 8-12 UserComSvr 类型库设计

值得一提的是，Delphi 在设计 TClientDataSet 组件和 TDataSetProvider 组件数据封包 Data 和 Delta 时就已经使用了 Variant 类型。Variant 是 Delphi 对 COM 最早的支持形式。从 Delphi 2 开始就已经有了该类型。Variant 和 OleVariant 的惟一区别在于前者支持所有类型，后者只支持在 Automation 中兼容的类型。比如指定一个 Pascal 字符串到一个 OleVariant，则它会自动转换为一个与 Automation 类型兼容的 BSTR 类型。

示例程序 8-8 COM 向导创建的接口单元 uUsers 的源代码

```
unit uUsers;

{$WARN SYMBOL _ PLATFORM OFF}

interface

uses
  Windows, ActiveX, Classes, ComObj, UserComSvr _ TLB, StdVcl,
  Variants;

type
  TUser = class (TTypedComObject, IUser)
  protected
    procedure getDepList (out Deps: OleVariant); stdcall;
    procedure getUserList (const strName: WideString; out vData: OleVariant);
      stdcall;
    procedure UpdateUserData (var vData: OleVariant; out nErr: SYSINT); stdcall;
    {Declare IUser methods here}
  end;

implementation
```

```

uses ComServ, uUserMaint;

procedure TUser.getDepList (out Deps: OleVariant);
var
  UserMaint: TUserMaint;
  DepLst: TStrings;
  i: Integer;
begin
  UserMaint: = TUserMaint.Create;
  try
    DepLst: = UserMaint.GetDepList;
    Deps: = VarArrayCreate ( [0, (DepLst.Count-1)] , varVariant );
    for i: = 0 to DepLst.Count-1 do
      begin
        Deps [i]: = DepLst.Strings [i];
      end;
    finally
      UserMaint.Free;
    end;
  end;

procedure TUser.getUserList (const strName: WideString;
  out vData: OleVariant);
var
  UserMaint: TUserMaint;
begin
  UserMaint: = TUserMaint.Create;
  try
    vData: = UserMaint.GetUserList (strName);
  finally
    UserMaint.Free;
  end;
end;

procedure TUser.UpdateUserData (var vData: OleVariant; out nErr: SYSINT);
var
  UserMaint: TUserMaint;
begin
  UserMaint: = TUserMaint.Create;
  try
    UserMaint.UpdateUserData (vData, nErr);
  finally
    UserMaint.Free;
  end;
end;

initialization
  TTypedComObjectFactory.Create (ComServer, TUser, Class _ User,
    ciMultiInstance, tmApartment);
end.

```

由于我们在示例程序 8-8 中完成了数据类型转换和接口实现功能，因此并不需要对封装的 `uUserMaint` 和 `udmUser` 单元做任何代码上的改动。这也体现了 COM/COM+ 封装技术的实用性。

这同时还启发我们考虑对一些原有的老程序进行 COM/COM+ 改造，以延长系统的生命期。

我们使用 Delphi 的 Run|Install COM+ Objects 菜单项，安装刚刚创建的 COM+ 组件。然后就可以修改和调试界面程序代码了。

示例程序 8-9 是空间分离后的界面单元代码。这里除了一些参数类型的转化以外也没有什么代码改动。

示例程序 8-9 空间分离后的界面单元代码

```
unit ufrmUsers;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBClient, StdCtrls, DBCtrls, Grids, DBGrids, Mask, ExtCtrls,
  Buttons, UserComSvr_TLB;

type
  TfrmUsers = class (TForm)
    btnExit: TButton;
    btnQryByName: TSpeedButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    edtQryByName: TLabeledEdit;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
    DBEdit4: TDBEdit;
    DBGrid1: TDBGrid;
    dbcbSex: TDBComboBox;
    dbcbDep: TDBComboBox;
    DataSource1: TDataSource;
    cdsUserMaint: TClientDataSet;
    cdsUserMaintID: TWideStringField;
    cdsUserMaintNAME: TWideStringField;
    cdsUserMaintSEX: TWideStringField;
    cdsUserMaintJOB: TWideStringField;
    cdsUserMaintTEL: TWideStringField;
    cdsUserMaintCALL: TWideStringField;
    cdsUserMaintDEP: TWideStringField;
    cdsUserMaintGROUP_ID: TWideStringField;
    cdsUserMaintPASSWORD: TWideStringField;
    btnUpdate: TBitBtn;
    procedure btnUpdateClick (Sender: TObject);
    procedure btnQryByNameClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
    procedure btnExitClick (Sender: TObject);
    procedure FormDestroy (Sender: TObject);
  end;

implementation
```

```

private
    objUsers: IUser;
public
    {Public declarations }
end;

var
    frmUsers: TfrmUsers;

const
    M_TITLE = '操作提示'; //所有提示对话框的标题

implementation

{$R *.dfm}

procedure TfrmUsers.btnUpdateClick (Sender: TObject);
var
    vDelta: OleVariant;
    nErr: integer;
begin
    if cdsUserMaint.State = dsEdit then cdsUserMaint.Post;
    if (cdsUserMaint.ChangeCount > 0) then
    begin
        vDelta: = cdsUserMaint.Delta;
        objUsers.UpdateUserData (vDelta, nErr);
        if nErr > 0 then
            application.MessageBox ('更新失败!', M_TITLE, MB_ICONWARNING)
        else
        begin
            application.MessageBox ('更新成功!', M_TITLE, MB_ICONINFORMATION);
            btnQryByNameClick (nil);
        end;
    end;
end;

procedure TfrmUsers.btnQryByNameClick (Sender: TObject);
var
    vData: OleVariant;
begin
    btnUpdate.Enabled: = true;
    cdsUserMaint.Active: = false;
    objUsers.GetUserList (widestring (edtQryByName.Text), vData);
    cdsUserMaint.Data: = vData;
    cdsUserMaint.Active: = True;
end;

procedure TfrmUsers.FormCreate (Sender: TObject);
var
    vDepList: OleVariant;
    i: integer;
begin
    objUsers: = CoUser.Create; //创建本地 COM+ 组件对象
    //objUsers: = CoUser.CreateRemote ('newdream'); //创建并连接远程 COM+ 组件对象

```

```
objUsers.GetDepList (vDepList);  
if (VarIsArray (vDepList)) and (not VarIsNull (vDepList)) then  
begin  
  for i : = VarArrayLowBound (vDepList, 1)  
    to VarArrayHighBound (vDepList, 1) do  
  begin  
    dbcbDep.Items.Add (vDepList [i]);  
  end;  
end;  
end;  
  
procedure TfrmUsers.btnExitClick (Sender: TObject);  
begin  
  close;  
end;  
  
procedure TfrmUsers.FormDestroy (Sender: TObject);  
begin  
  objUsers := nil;  
end;  
  
end.
```

对于 COM+ 组件, 读者可能更感兴趣如何在网络上不同的计算机之间进行调用, 这样封装的对象就可以实现分布式计算了。这里我没有使用 DCOMConnection 组件来可视化地进行远程连接。而是将应用程序中 COM+ 对象的创建方法由 Create 改为 CreateRemote (MachineName)。其中 MachineName 参数是 COM+ 对象所在远程机器的名称。比如在示例程序 8-9 中, 我们可以将:

```
objUsers := CoUser.Create;
```

改为:

```
objUsers := CoUser.CreateRemote ('newdream');
```

这样应用程序就可以调用在 newdream 计算机上的 COM+ 对象了。图 8-13 显示了客户端应用程序调用远程计算机上 COM+ 组件的情景。旋转的 COM+ 小球表示负责业务逻辑和数据库操作的服务器端 COM+ 组件正在工作中。客户端运行的瘦客户机应用程序也显示出数据库访问正常。

需要说明的是, 这里我只是想通过一个简单的数据库应用程序来演示由封装实现界面和业务分离的过程。该示例同时还说明了界面和业务分离由逻辑分离、物理分离到空间分离的进一步演化。这种演化既是软件开发的一种客观趋势, 也是不少程序员实际开发中的困惑所在。

这里没有深入探讨 COM+ 系统和数据库应用的开发, 也没有使用到 COM+ 的事务、安全和配置管理。严格意义上讲, 对于一个真正的数据库应用程序, 更新数据时必须要有事务 (transaction) 支持。Delphi 提供的 Transactional Object 和 Transactional Data Module 可以解决事务多段提交的问题。有兴趣的读者可以进一步查阅有关资料。

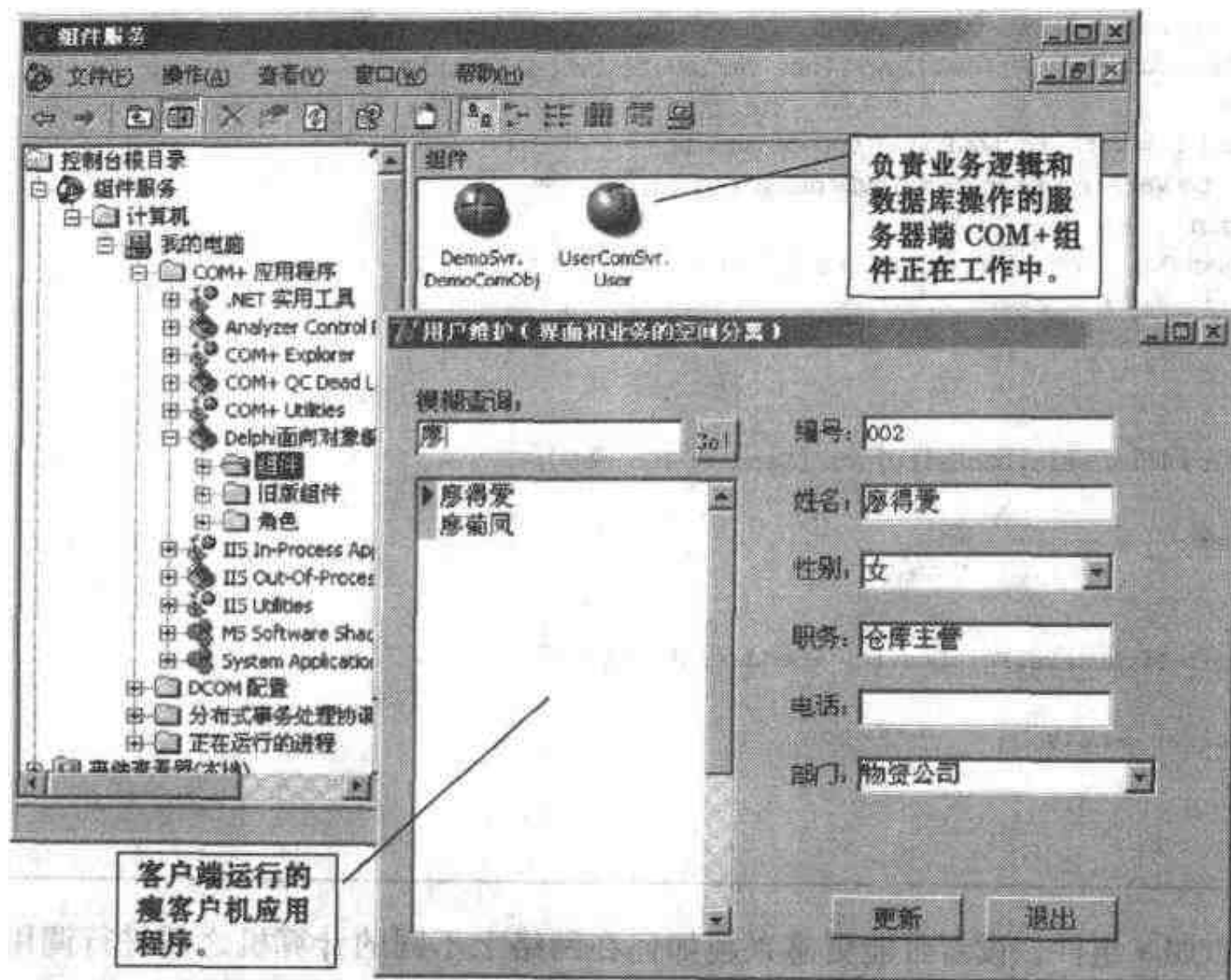


图 8-13 客户端应用程序调用远程计算机上的 COM+ 组件正在工作中

8.3 Web Service: 实现业务跨平台

在大型企业级应用中, 信息系统的开发是以业务为中心的。然而, 随着企业业务的不断增长以及来自不同厂商技术解决方案的大量使用, 往往会出现不同的操作系统和不同的应用平台, 这就使得新旧系统的通信和各种业务的整合变得日益困难。

令人振奋的是, 随着 Internet/Intranet 的飞速发展, XML 标准的成熟, 我们终于可以使用 Web Service (Web 服务) 来构建跨平台跨系统的业务服务了。而且还可以通过 Web Service 对一些原有的对象或组件进行再封装, 使得新旧系统的业务整合成本降至最低。

实际上, 从技术角度而言, 无论从哪个角度来看, Web Service 都是对象/组件技术在 Internet/Intranet 中的延伸。

8.3.1 Web Service 是一种部署在 Web 上的对象

从外部的使用者的角度而言, Web Service 是一种部署在 Web 上的对象/组件。它具备以下特征:

- 完好的封装性。Web Service 既然是一种部署在 Web 上的对象, 自然具备对象的良好封装性。对于使用者而言, 它能且仅能看到该对象提供的功能列表。
- 松散耦合。这一特征也是源于对象/组件技术。当一个 Web Service 的实现发生变更的时

候,调用者是不会感到这一点的。对于调用者来说,只要 Web Service 的调用界面不变,Web Service 的实现任何变更对他们来说都是透明的。甚至 Web Service 的实现平台从 COM+ 迁移到了 J2EE,或从 J2EE 迁移到了 .NET,用户都可以一无所知。对于松散耦合而言,尤其是在 Internet 环境下的 Web 服务而言,需要有一种适合 Internet 环境的消息交换协议。而 XML/SOAP 正是目前最为适合的消息交换协议。

- 使用协议的规范性。这一特征从对象而来,但与一般对象相比,其界面规范更加规范化和易于机器理解。首先,作为 Web Service,对象界面所提供的功能使用了标准的 WSDL 描述语言来描述;其次,由标准描述语言描述的服务界面应当是能够被发现的,因此这一描述文档需要被存储在私有的或公共的注册库里面。同时,使用标准描述语言描述的使用协议将不仅仅是服务界面,它还被延伸到 Web Service 的聚合、跨 Web Service 的事务、工作流等之中,而这些又都需要服务质量的保障。再次,安全机制对于松散耦合的对象环境十分重要,因此我们需要对诸如授权认证、数据完整性、消息源认证以及事务的不可否认性等运用规范的方法来描述、传输和交换。最后,在所有层次的处理都应当是可管理的,因此需要对管理协议运用同样的机制。
- 使用标准协议规范。Web Service,其所有公共的协议完全需要使用开放的标准协议进行描述、传输和交换。这些标准协议具有完全免费的规范,以便由任意厂商来实现。一般而言,绝大多数规范将由 W3C 或 OASIS 作为最终版本的发布方和维护方。
- 高度可集成能力。由于 Web Service 采取简单的、易理解的标准 Web 协议作为组件界面描述和协同描述规范,而完全屏蔽了不同软件平台的差异,无论是 CORBA、COM+ 还是 EJB 都可以通过这一种标准的协议进行互操作,实现了在当前环境下最高的可集成性。

实现一个完整的 Web Service 体系需要有一系列的协议和规范来支撑。这些协议和规范包括:

- 先前已经定义好的并且广泛使用的传输层和网络层的标准:IP、HTTP、SMTP 等。
- 目前开发的 Web Service 的相关标准协议,包括服务调用协议 SOAP、服务描述协议 WSDL 和服务发现/集成协议 UDDI,以及服务 workflow 描述语言 WSFL。
- 更高层的待开发的关于路由、可靠性以及事务等方面的协议。

Web Service 追求的第一目标是简单性。可能大家会觉得很奇怪,有那么多协议,怎么能说它简单呢?

首先,这些协议本身都是简单的,无论是 HTTP、FTP 等传统的 TCP/IP 系统的网络协议,还是 SOAP、WSDL、UDDI、WSFL 等基于 XML 的协议,其设计原则中极为重要的一点就是力求简单性。随着进一步深入了解 XML、SOAP 等协议,读者一定会深深体会到这一点。

其次,一个可以使用的 Web Service 应当按照需要选用若干层次的功能,而无需所有的特性。比如在目前状况下,一个简单应用可能只要使用 WSDL/SOAP 就可以架构一个符合规范的 Web Service 了。Delphi 中提供的 Web Service 支持正是建立在这样的构架之上。

最后,所有的机制完全是基于现有的技术,并没有创造一个完全的新体系。无论是 HTTP、FTP 这些现有的网络协议,还是 SOAP、WSDL 等这些基于 XML 而定义的协议都是遵循着一个

原则：继承原有的被广泛接受的技术。这样才能使得 Web Service 方便易用，容易推广。

Web Service 作为分布式应用的一种服务支持，特别是在实现业务跨平台方面有着强大的优势。Web Service 通过对各种业务对象的封装，包括对原有的已经封装了业务对象的各种组件的再次封装（如图 8-14 所示），最终可以将各种平台上的业务进行整合，提供统一的服务。这就是 Web Service 实现业务跨平台解决方案。

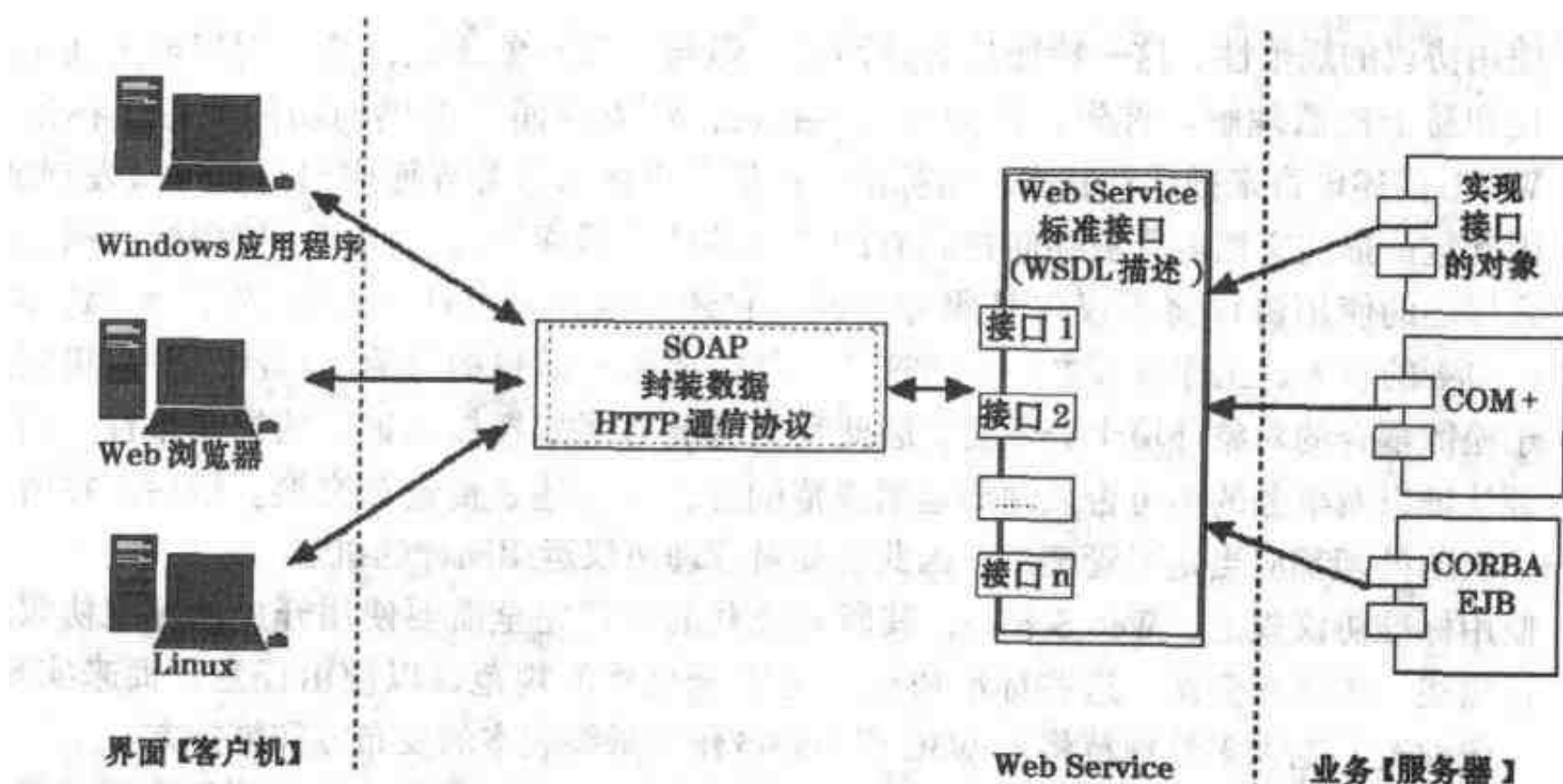


图 8-14 Web Service 实现业务跨平台示意图

8.3.2 创建 SOAP Server 应用程序

Web Service 为程序提供了通过 Internet 使用 SOAP 协议进行通信的机制。从概念上讲，这类似于微软的 COM+ (DCOM)，因为它支持分布式环境，但 Web Service 的用途更为广泛，即客户使用提供 Web Service 的 SOAP 应用程序时，不再需要关心其调用目标使用的是什么技术了。

SOAP 是 Simple Object Access Protocol 的缩写，是一个组件通过 HTTP 进行通信的、基于 XML 的标准。因为该协议是基于 XML 的，所以它是基于文本的，可以通过防火墙。

SOAP 应用程序将调用的目标（Delphi 对象、COM+ 组件或 EJB 等）封装成 Web 对象，以 Web Service 的形式提供给客户。

所以要实现 Web Service，在 Delphi 中首先要创建一个 SOAP 应用程序，然后定义一个或多个服务接口，封装对象或组件。

为了能够输出服务，还需要导出 WSDL。WSDL 是一个由 XML 组成的文件。它描述了 Web Service 如何初始建立与客户端的通信。通信的第一阶段称为查找阶段，在该阶段中，Web Service 传输可用的对象/组件接口信息。在完成了查找阶段后，就进入设置一般对象/组件接口调用的阶段。这些调用的声明和操作行为都非常类似于本地对象/组件，于是任何客户端或实现程序代码都可以通过这个接口获取服务。

下面我就来创建一个提供 Web Service 的 SOAP 应用程序示例。

首先在 Delphi 中的主菜单中选择 File|New|Other 菜单项，在弹出的 New Items 对话框中，找到 WebServices 选项页，选择 SOAP Server Application 图标，如图 8-15 所示。

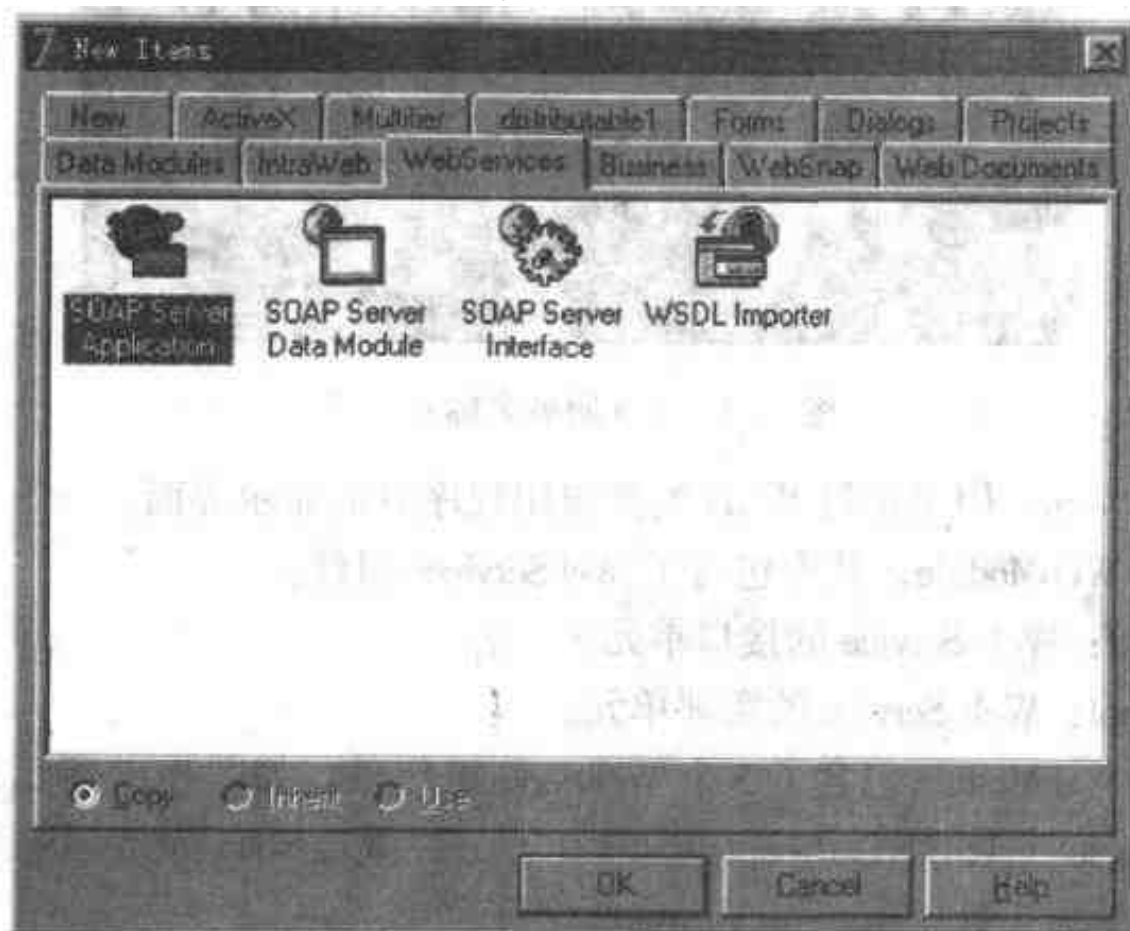


图 8-15 选择 SOAP Server Application 图标，新建 SOAP 应用程序

此时会接着弹出如图 8-16 所示的对话框。在该对话框中，我们可以选择新建 SOAP 应用程序类型。为了调试方便，在此先选择 Web App Debugger executable（简称 WAD）类型，待程序调试通过后再将其转换为实际使用的 CGI 类型（后面会介绍如何转换）。

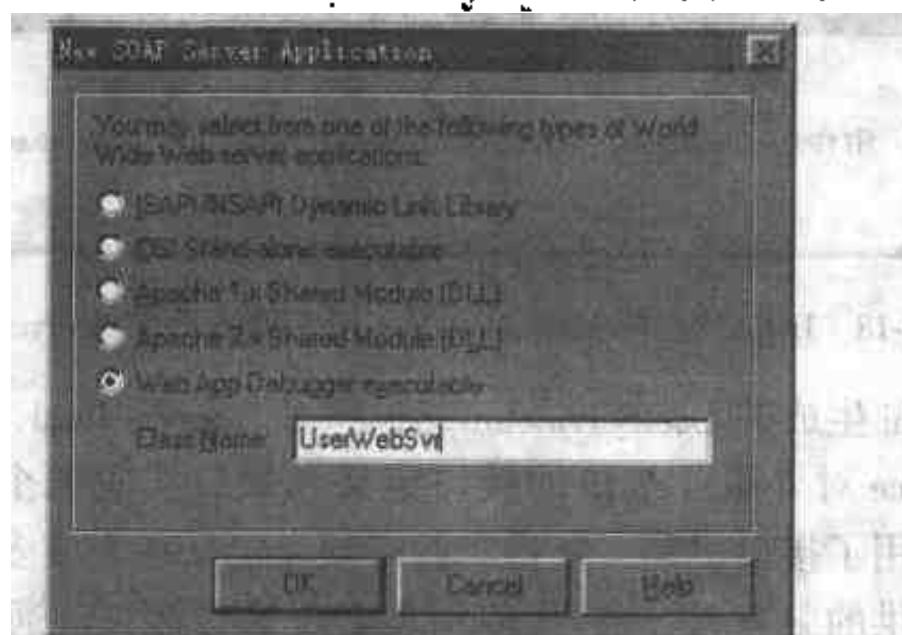


图 8-16 选择新建 SOAP 应用程序类型

由于这里选择的是 WAD 类型的应用程序，它实质上是一个 COM 对象，因此要在对话框的 Class Name 中输入一个名称。这样做的目的是为了解决 Web 应用程序不易调试的难题。

在随后弹出的 Add New WebService 对话框中，我们可以定义 Web Service 名称和使用单元，如图 8-17 所示。之后点击 OK 按钮，Delphi 会为我们创建一个 Web Service 项目，其中包含了以下文件：

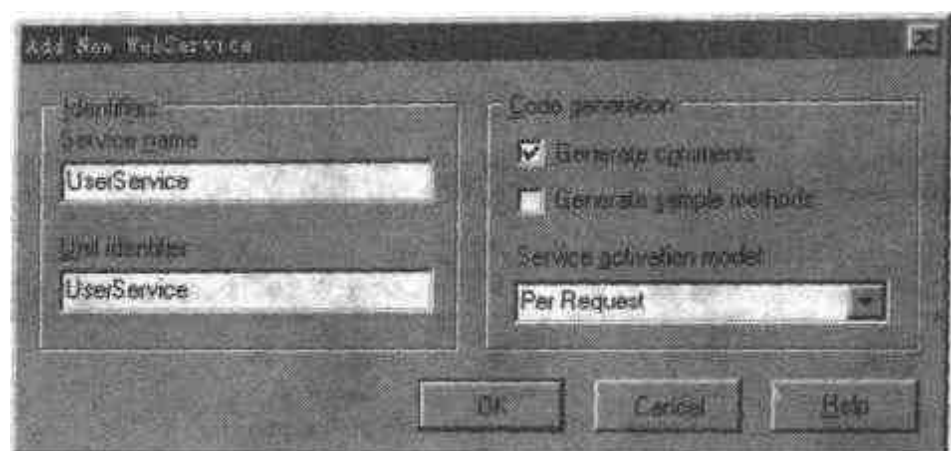


图 8-17 定义服务名称和单元

- Unit1: 一个 form, 用于运行 WAD 类型应用程序时的显示界面。
- Unit2: 一个 WebModule, 其中包含了 WebServices 组件。
- UserServiceIntf: Web Service 的接口单元。
- UserServiceImpl: Web Service 的实现单元。

其中 Unit2 中的 WebModule 包含了 3 个 WebServices 组件, 如图 8-18 所示, 它们分别是:

- HTTPSoapDispatcher1: THTTPSoapDispatcher ——拦截以 soap 为结尾的 post 请求, 将客户端的请求分派到 THTTPSoapPascalInvoker 组件。
- HTTPSoapPascalInvoker1: THTTPSoapPascalInvoker ——根据 THTTPSoapDispatcher 组件分派的请求, 调用正确的 Web Service 的实现方法来处理请求。
- WSDLHTMLPublish1: TWSDLHTMLPublish ——自动产生描述 Web Service 的服务信息并返回 WSDL 文件内容。



图 8-18 Delphi 创建的 WebModule, 其中包含了 WebServices 组件

我们仔细查看 Delphi 生成的 UserServiceIntf.pas 和 UserServiceImpl.pas 两个文件, 可以发现它们分别是提供 Web Service 对外服务接口的接口定义和接口实现两个不同单元。Delphi 在实现 Web Service 的服务时使用了接口机制, 使得我们必须先定义一个或多个接口, 并在接口中定义好方法 (Web Service 提供的服务), 最后再使用类实现这些接口。通过接口来将服务和实现服务的具体代码分离体现了面向对象编程的思想, 保证了 Web Service 所提供服务的可维护性和可扩展性, 并为我们进一步封装业务对象提供了方便。

8.3.3 用 Web Service 封装业务对象

对于分布式系统而言, 在远程对象上调用方法的一个更简单的方法是把这些对象放在服务程序中。Web Service 是一个实现分布式系统的重要概念, 它可以从根本上改变设计应用程序的方式。在离散的、受控制的服务器组上运行应用程序, 将让位给在广泛分布的资源集上包装应

用程序、处理和显示数据。所以用 Web Service 来封装业务对象，可以让这些业务资源分布在不同类型的不同服务器和操作系统上，通过 Web Service 的通用接口，我们处理这些来自不同地方的业务对象，就像是处理在同一个系统上的业务对象一样。

实际上，真正的 Web Service 是用于业务层的服务，没有用户界面。它主要用于程序和程序之间的通信。Delphi 在创建 WAD 类型的 SOAP 应用程序时提供了一个窗体界面，是为了让开发人员可以从一个 Web Service 上“窥视”结果，但这仅方便了开发。记住，Web Service 一般是与业务对象/组件对话，而不是与用户对话。

下面将结合示例程序介绍如何使用 Web Service 来封装业务对象。在 Delphi 中，Web Service 的编程实际上要比读者想像得更容易，因为 Web Service 编程非常类似于其他类型的对象编程。

大家还记得前面一节我们讨论过的“界面和业务分离的演化示例”吗？其中谈到了如何将界面和业务进行逻辑分离。我们在设计任何一个应用程序时都应该用这种思想去考虑问题，这会让我们受益匪浅，因为这样设计出的程序以后无论是维护还是升级都非常方便。现在就用 Web Service 来封装“界面和业务分离的演化示例”中的业务逻辑部分，这部分的源代码在示例程序 8-3 和示例程序 8-4 中。

我们可以创建一个项目组（ProjectGroup1），加入“界面和业务分离的演化示例”中的 distributable1 项目和刚才创建的 Web Services 项目（重新命名为 WebService 项目）。将 distributable1 项目中原来的业务逻辑单元 uUserMaint 和 udmUser 移动到 WebService 项目中，如图 8-19 所示。这是界面和业务由逻辑分离迈向物理分离和空间分离的第一步。

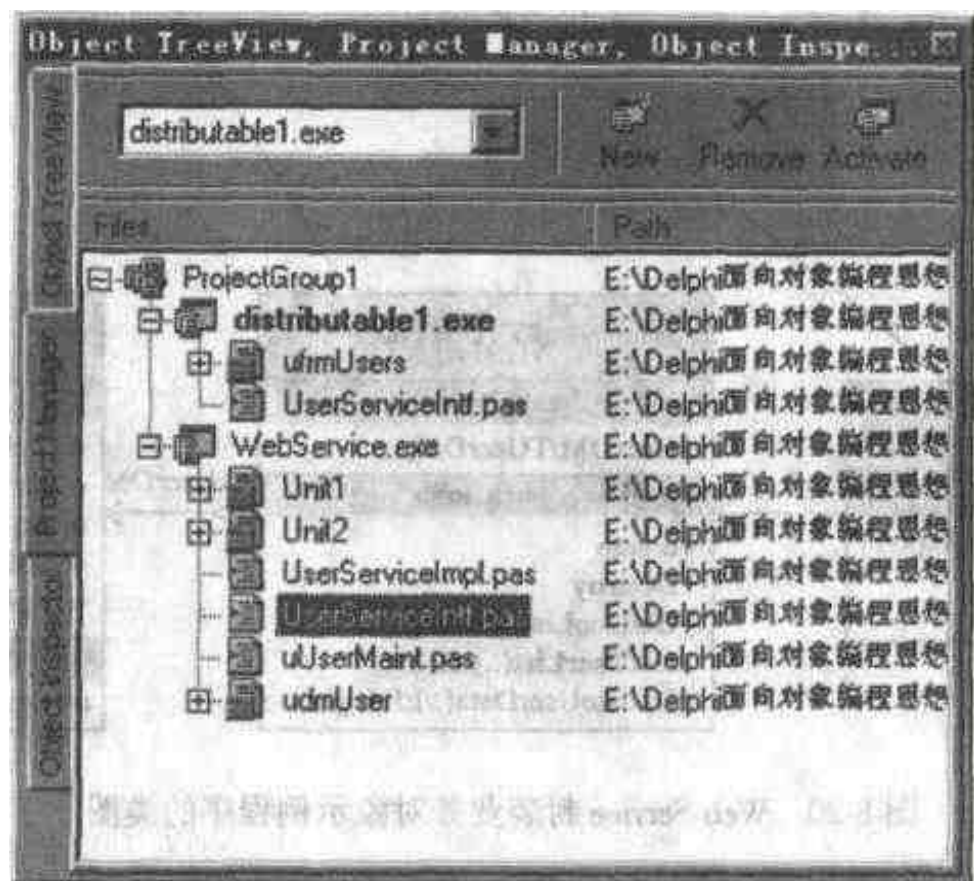


图 8-19 为 Web Service 封装业务对象示例程序创建一个项目组

接下来，要在 UserServiceIntf.pas 和 UserServiceImpl.pas 这两个单元文件中创建接口和接口的实现代码。提供 Web Service 的接口包含有：GetDepList、GetUserList 和 UpdateUserData 三个方法。记住，不能简单地把 uUserMaint.pas 中的代码复制到 UserServiceImpl.pas 中。因为 Web Service 不


```

uses InvokeRegistry, Types, XSBuiltIns;

type

  {Invokable interfaces must derive from IInvokable}
  IUserService = interface (IInvokable)
    ['{1FAB59BF-83FD-4855-835C-AB392086FB83}']
    function GetDepList (out iCount: integer): TStringDynArray; stdcall;
    function GetUserList (strName: String): String; stdcall;
    function UpdateUserData (UserData: String): integer; stdcall;
  end;

implementation

initialization
  {Invokable interfaces must be registered}
  InvRegistry.RegisterInterface (TypeInfo (IUserService));

end.

```

由于 Web Service 采用了 SOAP 协议来封装数据类型，可以定义比 XML 数据类型更细致的规则，因此适合远程系统交换数据。加上 SOAP 是建立在 XML 之上的，这使得各种不同的操作系统和应用平台都能够用相同的规则交换和封装数据，其打破了异构系统间的限制，实现了跨平台的应用。

然而 SOAP 使用的数据类型毕竟不同于 Delphi 的数据类型。好在对于简单的数据类型，Delphi 能够根据 SOAP 规范自动完成数据的封装和解析。但对于复杂的数据类型则需要程序员进一步编程解决。

读者阅读示例程序 8-11 可以发现，这里所做的主要工作是完成数据类型的转换。在 GetDepList 函数中，我们将 TUserMaint 类的 GetDepList 函数返回值数据类型 TStrings 转换成了 SOAP 兼容的 TStringDynArray 数据类型，TStringDynArray 是字符串类型的动态数组。它由系统的 Types 单元提供，所以要在程序的 interface 部分使用该单元（需要手工加入）。在客户端应用程序中，我们可以将该动态数组再转换成 TStrings 类型。

示例程序 8-11 IUserService 接口的实现单元 (UserServiceImp.pas I)

```

{Invokable implementation File for TUserService which implements IUserService}

unit UserServiceImpl;

interface

uses InvokeRegistry, classes, Types, XSBuiltIns, uUserMaint, UserServiceIntf;

type

  {TUserService}
  TUserService = class (TInvokableClass, IUserService)
  public

```

```

    function GetDepList (out iCount: integer): TStringDynArray; stdcall;
    function GetUserList (strName: String): String; stdcall;
    function UpdateUserData (UserData: String): integer; stdcall;
end;

implementation

function TUserService.GetDepList (out iCount: integer): TStringDynArray;
var
    ObjUser: TUserMaint;
    i: integer;
    tmpstrs: TStringList;
begin
    tmpstrs := TStringList.Create;
    ObjUser := TUserMaint.Create;
    tmpstrs.Assign (ObjUser.GetDepList);
    iCount := tmpstrs.Count;
    SetLength (result, iCount);
    for i := 0 to iCount-1 do
        Result [i] := tmpstrs.Strings [i];
    end;
    tmpstrs.Free;
    ObjUser.Free;
end;

function TUserService.GetUserList (strName: String): String;
var
    ObjUser: TUserMaint;
begin
    ObjUser := TUserMaint.Create;
    try
        result := ObjUser.GetUserList (strName);
    finally
        ObjUser.Free;
    end;
end;

function TUserService.UpdateUserData (UserData: String): integer;
var
    ObjUser: TUserMaint;
begin
    ObjUser := TUserMaint.Create;
    try
        result := ObjUser.UpdateUserData (UserData);
    finally
        ObjUser.Free;
    end;
end;

initialization
    | Invokable classes must be registered |
    InvRegistry.RegisterInvokableClass (TUserService);
end.

```

在原来的示例程序 8-2 中，数据集的传递是通过 TClientDataSet 组件 cdsUserMaint 的数据封

包实现的。查询数据时，cdsUserMaint 从 objUsers 对象获得 Data 包；更新数据时，cdsUserMaint 将数据更新变化装在 Delta 包中传出，交给 TDataSetProvider 完成数据库更新。但是 Web Service 不支持 OleVariant 类型的数据封包。

好在 TClientDataSet 组件还提供了 XMLData 形式的数据封包，其数据类型是 SOAP 兼容的 String 类型，所以我们需要分别改动 TUserMaint 的 GetUserList 和 UpdateUserData 方法，以及数据模块单元 UserDM，但改动不是很大。

具体的方法是在数据模块 UserDM 中增加 TClientDataSet 组件 cdsUser，如图 8-21 所示。并将 cdsUser 的 ProviderName 属性设为 dspUser，由 cdsUser 负责转换和解析 XMLData 形式的数据封包。这同样需要对应修改客户端的应用程序，由客户端的 TClientDataSet 组件负责转换和解析 XMLData 形式的数据封包。改动后的 uUserMaint 单元程序源代码如示例程序 8-12 所示。

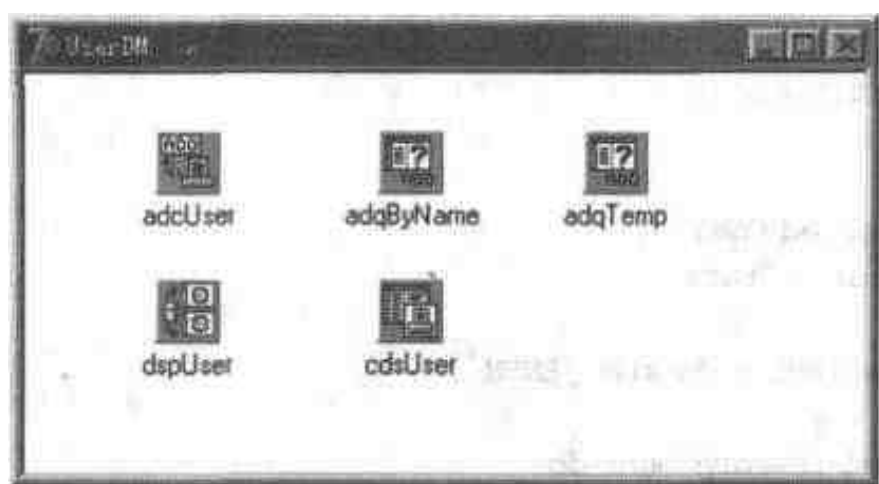


图 8-21 UserDM 中新增了 cdsUser 组件，
它负责转换和解析 XMLData 形式的数据封包

示例程序 8-12 改动后的程序可以利用 XMLData 形式的数据封包获取和更新数据

```
unit uUserMaint;
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, DBClient, udmUser;

type
  TUserMaint = class(TObject)
  public
    UserDM: TUserDM;
    function GetDepList: TStrings;
    function GetUserList (strName: String): String;
    function UpdateUserData (UserData: String): integer;
    constructor create;
    destructor Destroy; override;
  end;

implementation

;
x * * * * * TUserMaint
|
```

```

constructor TUserMaint.create;
begin
  UserDM: = TUserDM.Create (nil);
end;

destructor TUserMaint.Destroy;
begin
  freeandnil (UserDM);
  inherited;
end;

function TUserMaint.GetDepList: TStrings;
var
  i: Integer;
  tmpstrs: TStrings;
begin
  tmpstrs: = TStringlist.Create;
  with UserDM do
  try
    if not adcUser.Connected then
      adcUser.Connected: = True;
    adqTemp.sql.Clear;
    adqTemp.sql.add ('select * from M_BMBM');
    adqTemp.Open;
    for i: = 1 to adqTemp.RecordCount do
    begin
      tmpstrs.Add (adqTemp.Fieldbyname ('BMMC').AsString);
      adqTemp.Next;
    end;
    adqTemp.Close;
    result: = tmpstrs;
  finally
    adcUser.Connected: = False;
  end;
end;

function TUserMaint.GetUserList (strName: String): String;
begin
  with UserDM do
  try
    if not adcUser.Connected then
      adcUser.Connected: = True;
    with adqByName do
    begin
      close;
      Parameters.ParamByName ('name').value: = '%' + strName + '%';
      open;
      cdsUser.Active: = True;
      result: = cdsUser.XMLData;
    end;
  finally
    adcUser.Connected: = False;
  end;
  GetDepList;
end;

```

```
end;  
  
function TUserMaint.UpdateUserData (UserData: string): integer;  
begin  
    UserDM.cdsUser.XMLData: = UserData;  
    UserDM.cdsUser.Active: = true;  
    result: = UserDM.cdsUser.ApplyUpdates (-1);  
end;  
  
end.
```

保存并运行我们创建的这个 SOAP 应用程序，Delphi 便完成了该程序的编译和注册工作。选择 Delphi 菜单上的 Tools|Web App Debugger 菜单项，运行 Web App Debugger，如图 8-22 所示。点击 Start 按钮启动服务。如果点击 Default URL 链接，可以进一步查看 Web 服务器的信息。

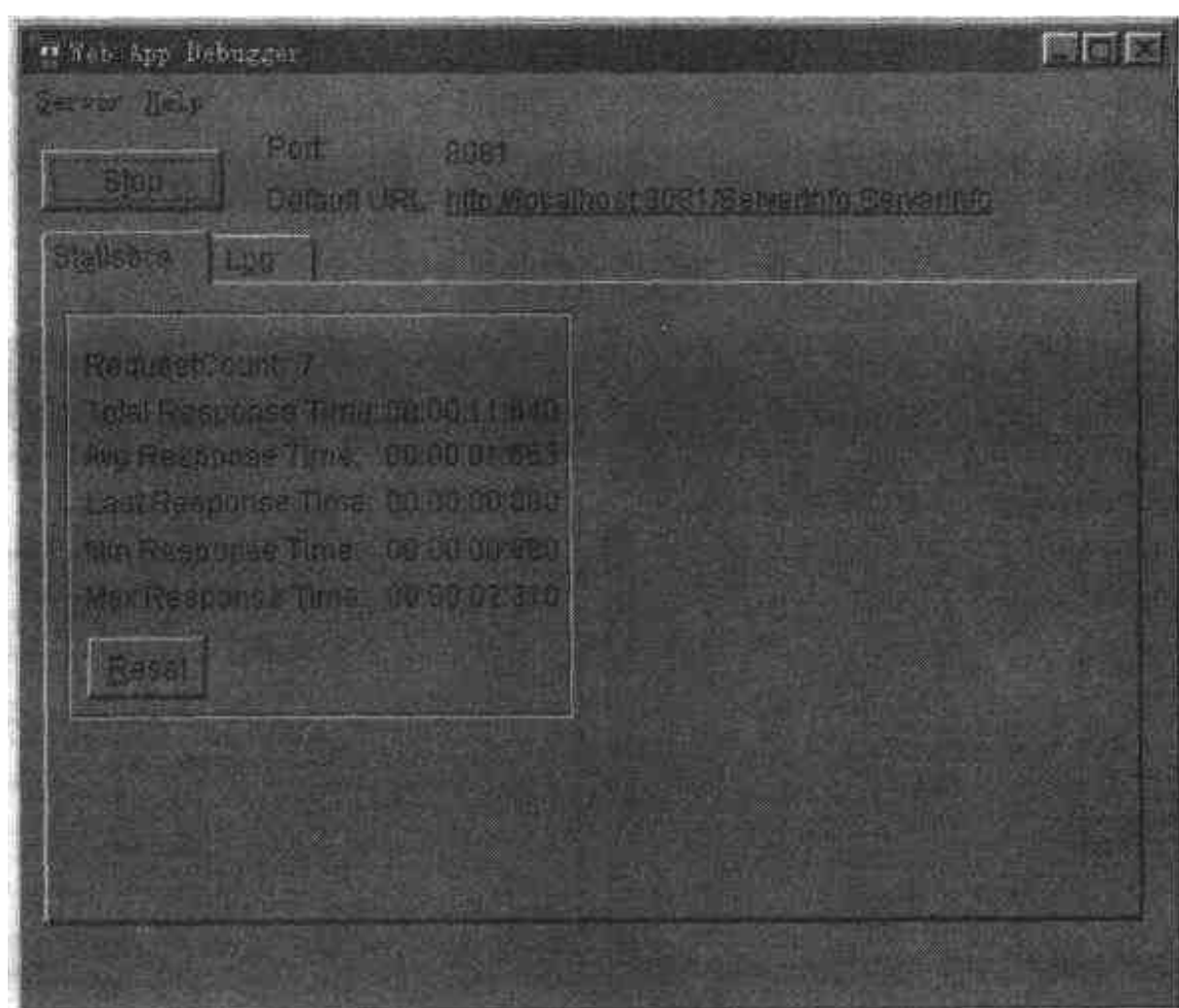


图 8-22 运行 Web App Debugger 可以调试 SOAP 应用程序

图 8-23 是点击 Default URL 链接后显示在网页上的已注册的服务器信息。注意，这里所能看见的是 WAD 类型的服务器，因为该类型的服务器本身就是一个已注册的 COM 服务器。

选择我们刚才创建的 WebService.UserWebSvr，单击 Go 按钮，可以进一步查看到该服务的输出接口信息及 WSDL，如图 8-24 所示。从图 8-24 中，我们可以看到该示例程序 Web Service 的 TUserService 接口及其所提供方法的描述，以及更详细的 WSDL 内容。

WSDL 包含了该 Web Service 的重要信息，比如 Port Name 的定义以及访问该 Web Service 的 URL 地址。我们在随后的客户端应用程序中需要用到这些信息，以便使应用程序能够绑定到该 Web Service 上。

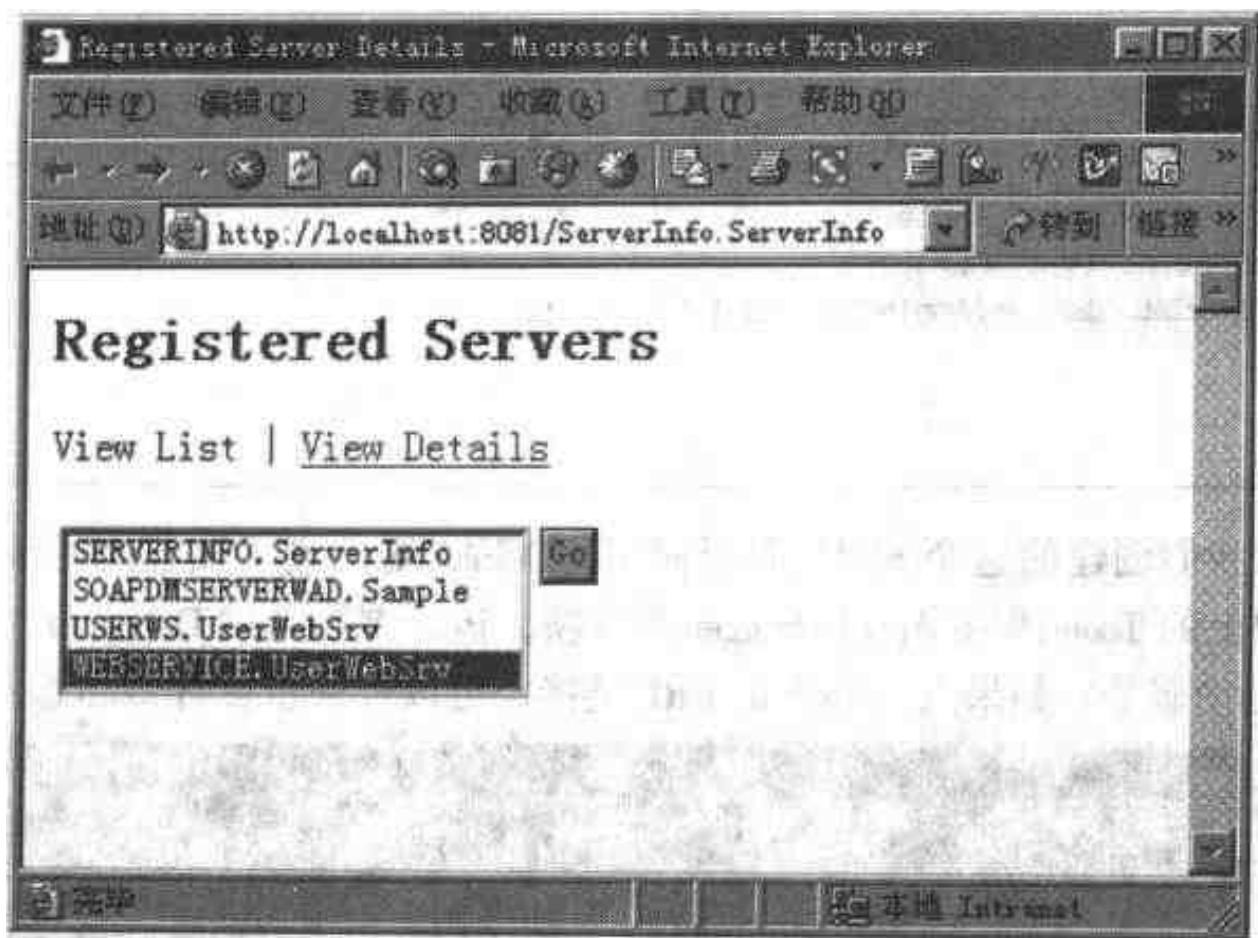


图 8-23 查看已注册的服务器信息

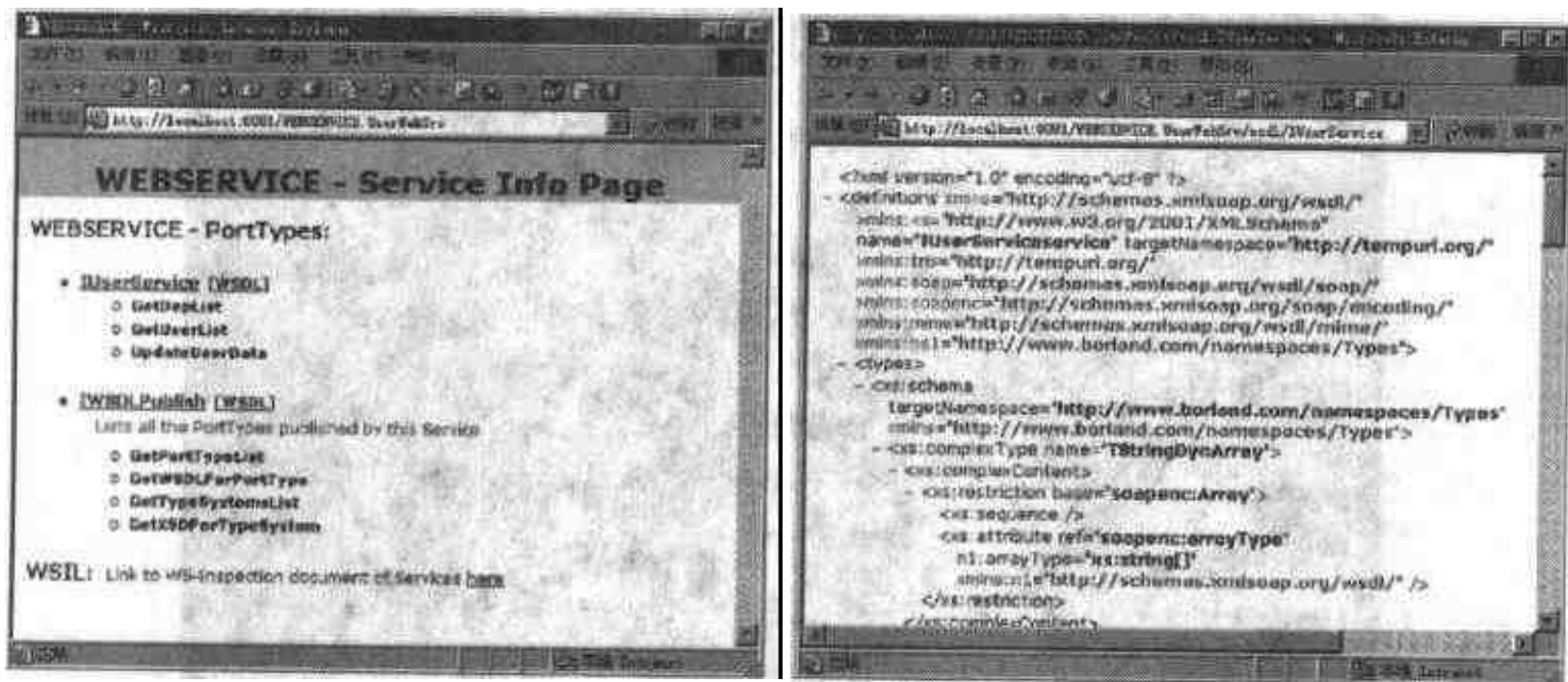


图 8-24 查看 Web Service 的输出接口信息及 WSDL

8.3.4 创建调用 Web Service 的客户端程序

调用 Web Service 的客户端应用程序可以是多种不同的形式，比如：Windows 可执行文件、Web 页面等。这里我们通过对原先的示例程序 8-2 进行稍微的改动来实现一个 Web Service 的客户端应用程序。为的是说明以面向对象思想来实现界面和业务分离编程所体现的优越性。

我们切换到图 8-19 所示的 distributable1 项目，打开 ufrmUsers 单元，在 frmUsers 窗体上新增 THHTTPRIO 组件 HTTPRIO1，如图 8-25 所示。THHTTPRIO 组件可以通过 HTTP 和 SOAP 数据封包调用远程 Web Service。接下来我们要为 HTTPRIO1 进一步设置属性，如图 8-26 所示。其中有关 Web Service 绑定信息的属性设置值来自图 8-24 所示的 WSDL：

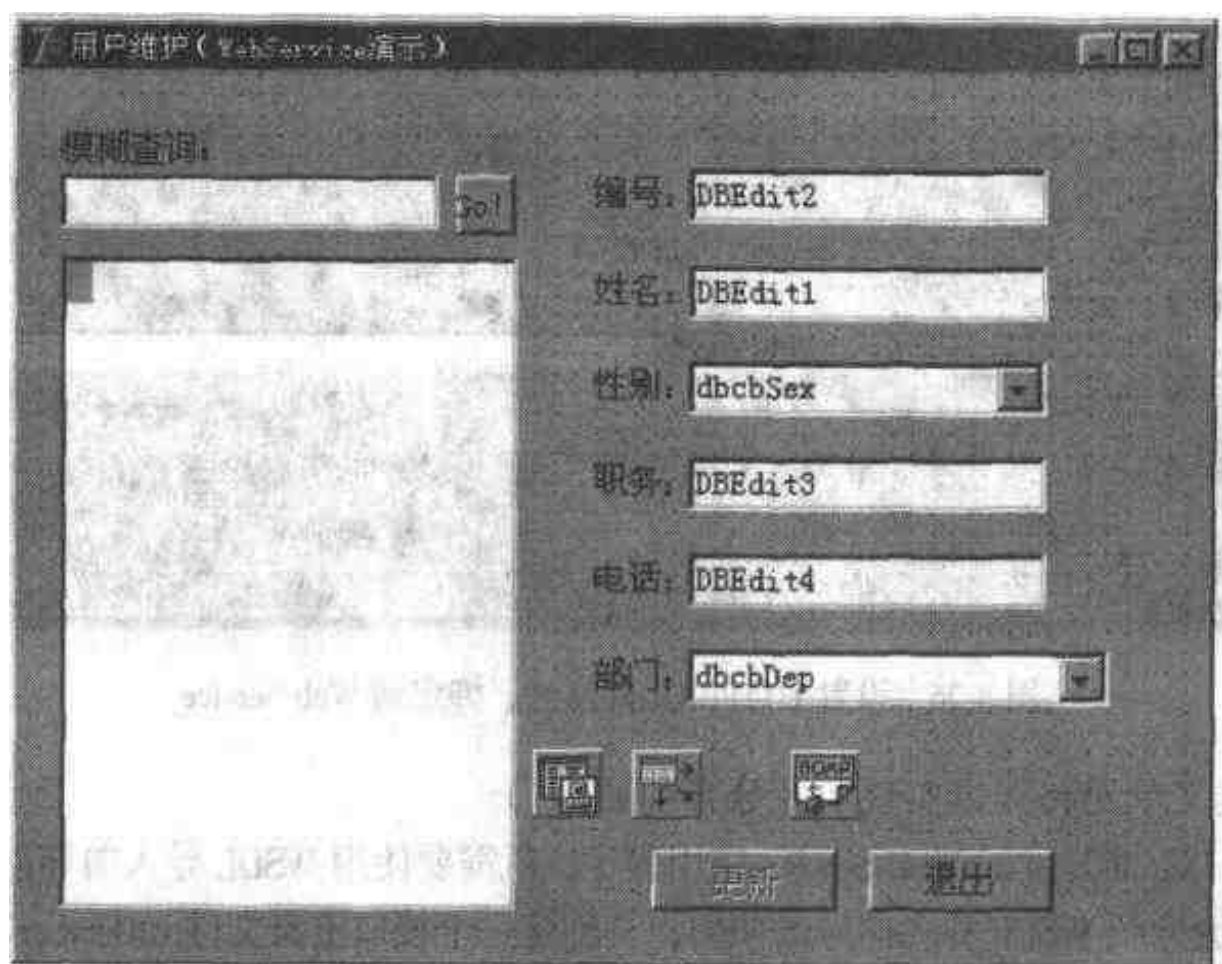


图 8-25 在 frmUsers 窗体上新增 THHTTPRIO 组件 HTTPRIO1

```

- < service name = " IUserServiceservice" >
- < port name = " IUserServicePort" binding = "tns: IUserServicebinding" >
  < soap: address
location = "http://localhost: 8081/WEBSERVICE.UserWebSrv/soap/IUserService"
  />
  </port>
</service>

```

THHTTPRIO 组件的 WSDLLocation 属性的格式为:

http://Web 服务器网址/Web Service 应用程序/wsd/接口名称

在我的机器上, 该属性值为:

http://localhost: 8081/WebService.exe/wsd/IUserService

由于我现在调试的是 WAD 类型的 Web Service 应用程序, 是以 COM 对象实现的, 因此也可以将属性值设为:

http://localhost: 8081/WebService.UserWebSrv/wsd/IUserService

然后可以接着设置 Port 和 Service 属性, 如图 8-26 所示。

示例程序 8-13 给出了调用 Web Service 的客户端应用程序源代码, 通过 HTTPRIO1 组件, 我们可以获得 Web Service 的接口, 调用 Web Service 的服务:

```
IUser: = (HTTPRIO1 as IUserService);
```

需要注意的是, 客户端应用程序还需要能够获得 Web Service 服务接口的 Object Pascal 定义。为此, 在图 8-19 所示的 distributable1 项目中, 加入了 ulUserService.pas 文件。对 Delphi 开发的 Web Service 应用程序, 可以方便地在客户端直接使用 Web Service 接口定义单元。(这也是为

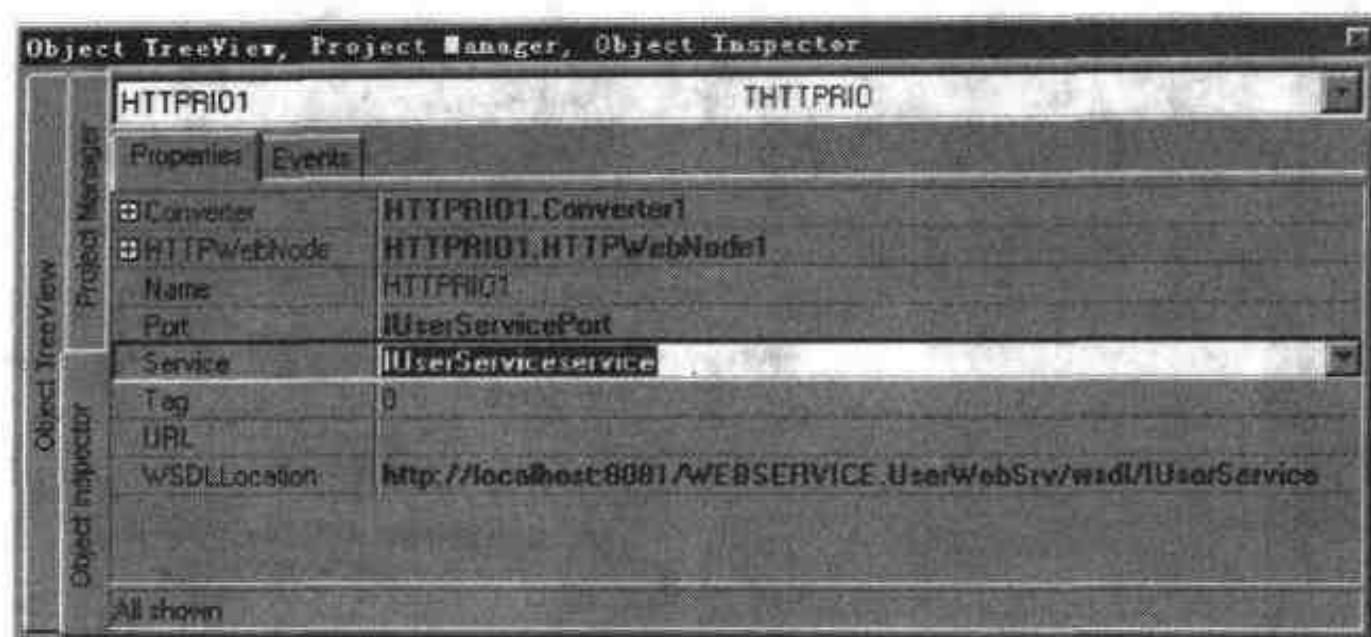


图 8-26 设置 HTTPRIO1 的属性，绑定到 Web Service

什么要将接口定义单元和接口实现单元分开的原因。)

如果不是用 Delphi 开发的 Web Service 应用程序，就需要使用 WSDL 导入向导创建的一个接口定义文件。后面我会介绍 WSDL 导入向导的使用，并创建一个接口定义文件 uUserService.pas。

示例程序 8-13 调用 Web Service 的客户端应用程序

```
unit ufrmUsers;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, DBClient, StdCtrls, DBCtrls, Grids, DBGrids, Mask, ExtCtrls,
  Buttons, InvokeRegistry, Rio, SOAPHTTPClient, Types;

type
  TfrmUsers = class (TForm)
    btnExit: TButton;
    btnQryByName: TSpeedButton;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    edtQryByName: TLabeledEdit;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
    DBEdit4: TDBEdit;
    DBGrid1: TDBGrid;
    dbcbSex: TDBComboBox;
    dbcbDep: TDBComboBox;
    DataSource1: TDataSource;
    cdsUserMaint: TClientDataSet;
    cdsUserMaintID: TWideStringField;
```

```

    cdsUserMaintNAME: TWideStringField;
    cdsUserMaintSEX: TWideStringField;
    cdsUserMaintJOB: TWideStringField;
    cdsUserMaintTEL: TWideStringField;
    cdsUserMaintCALL: TWideStringField;
    cdsUserMaintDEP: TWideStringField;
    cdsUserMaintGROUP_ID: TWideStringField;
    cdsUserMaintPASSWORD: TWideStringField;
    btnUpdate: TBitBtn;
    HTTPRIO1: THTTPRIO;
    procedure btnUpdateClick (Sender: TObject);
    procedure btnQryByNameClick (Sender: TObject);
    procedure btnExitClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
private
public
    |Public declarations |
end;

var
    frmUsers: TfrmUsers;

const
    M_TITLE = '操作提示'; //所有提示对话框的标题

implementation

uses uIUserService; //使用 WSDL 导入向导创建的 uIUserService.pas 文件
// uses UserServiceIntf; {对 Delphi 开发的 Web Service 应用程序, 可以直接使用 UserServiceIntf 单元!}

{$R *.dfm}

procedure TfrmUsers.btnUpdateClick (Sender: TObject);
var
    IUser: IUserService;
begin
    IUser: = (HTTPRIO1 as IUserService);
    if IUser.UpdateUserData (cdsUserMaint.XMLData) = 0 then
        application.MessageBox ('更新成功!', M_TITLE, MB_ICONINFORMATION)
    else
        application.MessageBox ('更新失败!', M_TITLE, MB_ICONSTOP);
    IUser: = nil;
end;

procedure TfrmUsers.btnQryByNameClick (Sender: TObject);
var
    IUser: IUserService;
begin
    IUser: = (HTTPRIO1 as IUserService);
    btnUpdate.Enabled: = true;
    cdsUserMaint.Active: = false;
    cdsUserMaint.XMLData: = IUser.GetUserList (edtQryByName.Text);
    cdsUserMaint.Active: = True;

```



```
IUser: = nil;
end;

procedure TfrmUsers.btnExitClick (Sender: TObject);
begin
  close;
end;

procedure TfrmUsers.FormCreate (Sender: TObject);
var
  IUser: IUserService;
  i, count: integer;
  aDeps: TStringDynArray;
begin
  count: = 0;
  IUser: = (HTTPRIO1 as IUserService);
  aDeps: = IUser.GetDepList (count);
  for i: = 0 to count-1 do
    dbcbDep.Items.Add (aDeps [i]);
  end;
end.

end.
```

编译并运行这个客户端程序，发现该程序自动激活了提供 Web Service 服务的服务端 SOAP 应用程序，可以完成 Web Service 服务的调用。显然，对于操作用户（End User）而言，使用这个调用 Web Service 的客户端程序与使用示例程序 8-1 或示例程序 8-2 没有什么区别。

8.3.5 Web Service 类型的转换和部署

Web Service 的成功运用的确给我们带来了不少好处。那么如何将 WAD 类型的 SOAP 应用程序转换为真正实用的其他类型呢？如何在 Web 服务器上部署和获取 Web Service 呢？

我还用刚才的示例来回答这些问题。下面先介绍如何把 WAD 类型的 SOAP 应用程序转换为常用的 CGI 类型的 SOAP 应用程序。

首先在 Delphi 的主菜单中选择 File|New|Other 菜单项，在弹出的 New Items 对话框中，找到 WebServices 选项页，选择 SOAP Server Application 图标，如图 8-15 所示。

接着在如图 8-16 所示的对话框中选择 CGI Stand-alone executable 类型。此时，Delphi 将创建一个新的 CGI 类型的 SOAP 应用程序项目。注意，在是否创建 SOAP 模块接口对话框中选择“No”，因为我们后面要使用原先在 WAD 类型程序中调试好的接口及接口实现代码单元。

将新建 CGI 类型的 SOAP 应用程序项目命名为 UserWS，将项目中新建的 WebModule 单元命名为 wmForm。并在项目中加入原来调试好的接口及接口实现代码单元文件：UserServiceIntf、UserServiceImpl、uUserMaint 和 udmUser，如图 8-27 所示。

编译该 CGI 类型的 SOAP 应用程序，但无需运行（第一，该应用程序不是 COM；第二，它没有界面，无需窗体）。编译无误后将其部署到 Web 服务器上。对于微软的 IIS，可以利用其虚拟目录创建向导，来发布我们刚刚创建的 CGI 类型的 SOAP 应用程序。对于 CGI 程序还要注意设置好足够的 Web 访问权限，如图 8-28 所示。最后，在 IIS 中，我们可以查看到设置好的虚拟



图 8-27 新建 CGI 类型的 SOAP 应用程序，并加入原来的文件

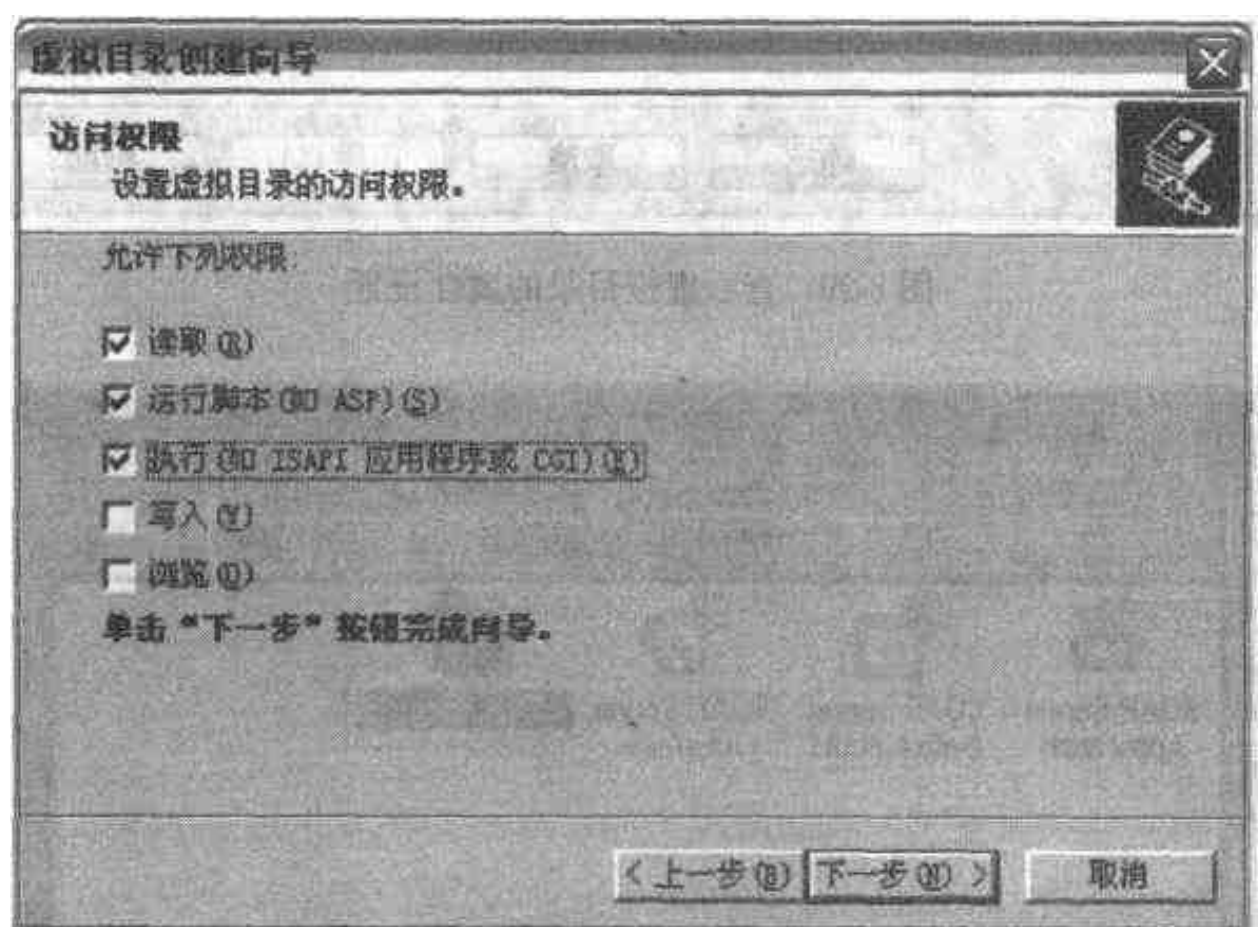


图 8-28 利用 IIS 虚拟目录创建向导来发布 CGI 类型的 SOAP 应用程序

目录属性，如图 8-29 所示。

虽然现在的 CGI 程序 UserWS.exe 可以提供 Web Service 服务了，但我们的客户端应用程序如何才能找到并绑定这个 Web Service 呢？如果这个 Web Service 不是我们自己用 Delphi 开发的，又如何获取它的 WSDL 呢？

我们可以利用 Delphi 的 WSDL 导入向导解决这些问题。如图 8-30 所示，选择 WSDL Importer，单击 OK，启动 WSDL 导入向导。

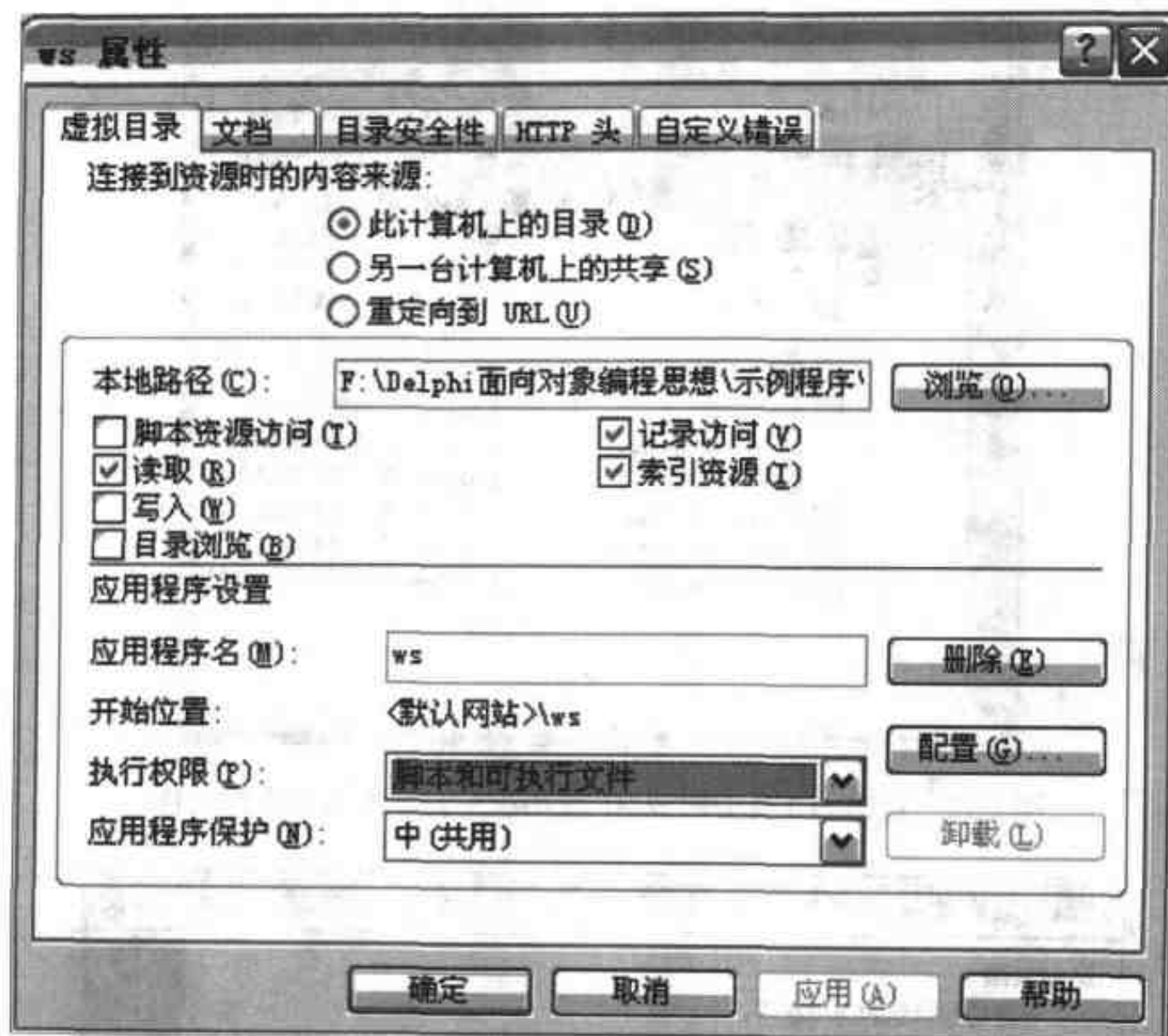


图 8-29 查看虚拟目录的属性设置

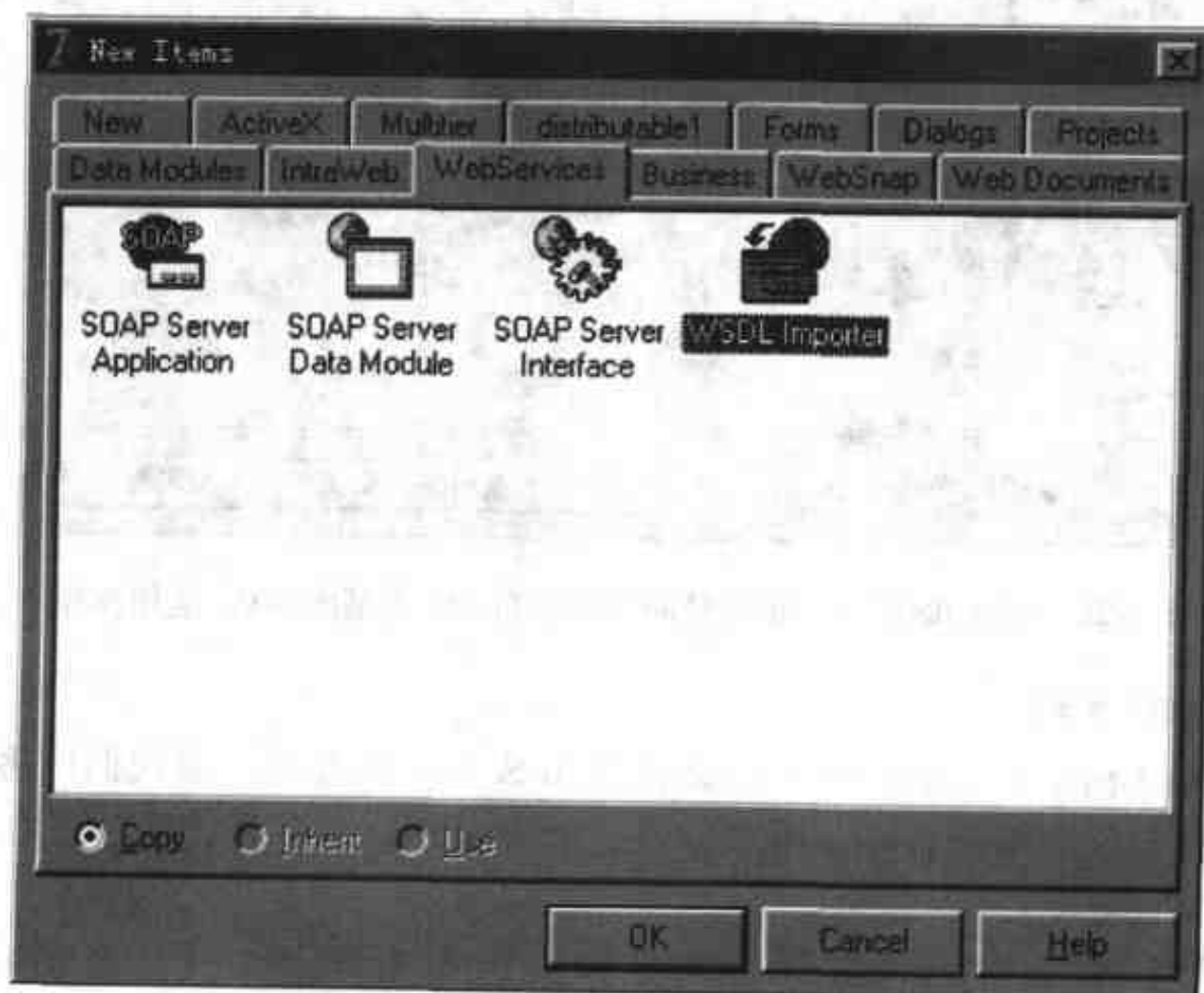


图 8-30 使用 WSDL 导入向导

接着在如图 8-31 所示的对话框中输入刚才创建的 CGI 程序的 Web 地址，单击 Next。向导会根据 Web Service 的 WSDL 自动创建一个符合 Delphi 要求的接口定义文件，如图 8-32 所示。阅读示例程序 8-14 所示的 Web Service 接口定义文件源代码，我们可以找到供客户端程序调用的接口方法，以及 THTTTPRIO 组件的属性设置值。

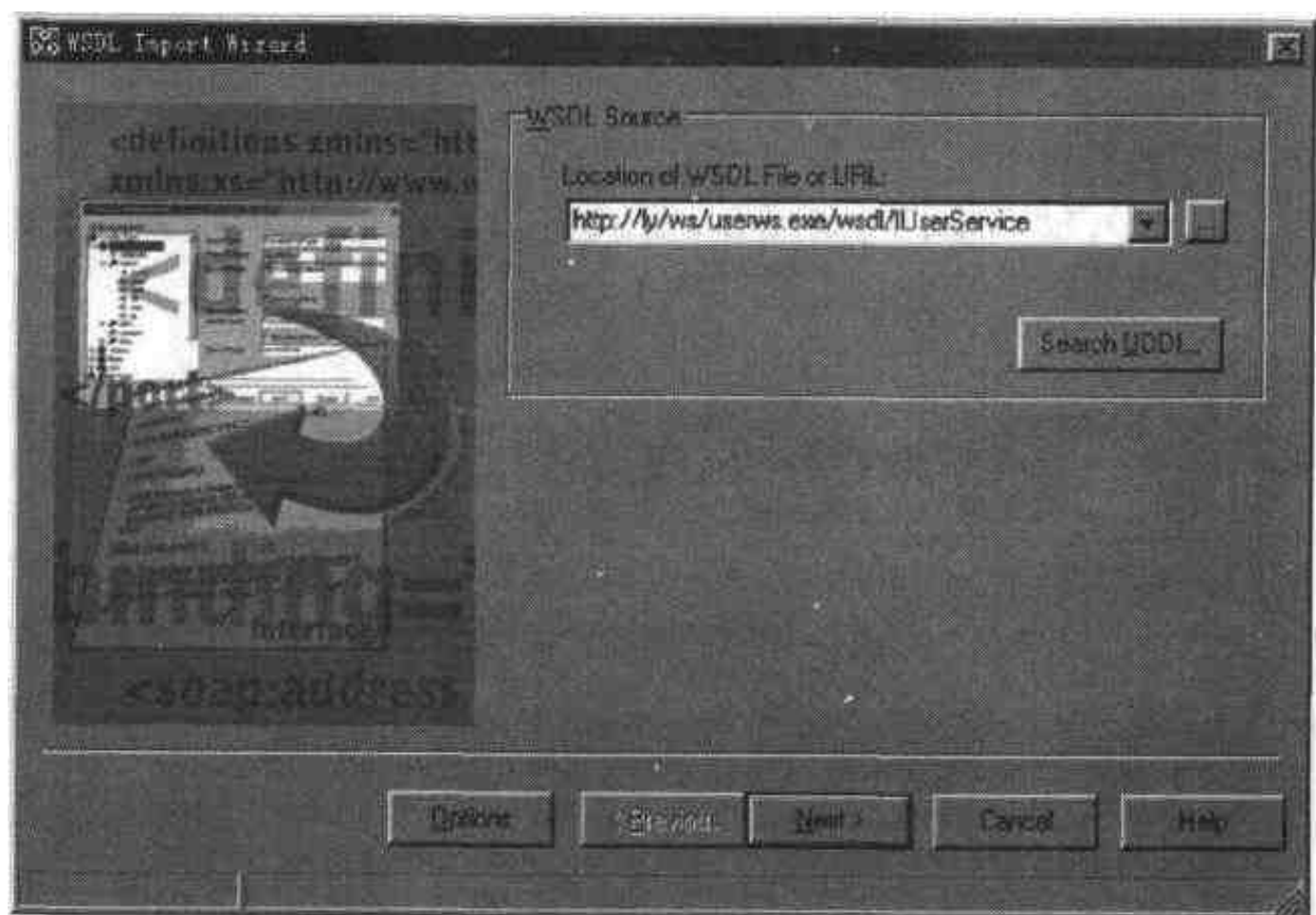


图 8-31 在对话框中输入 Web Service 的 WSDL 地址

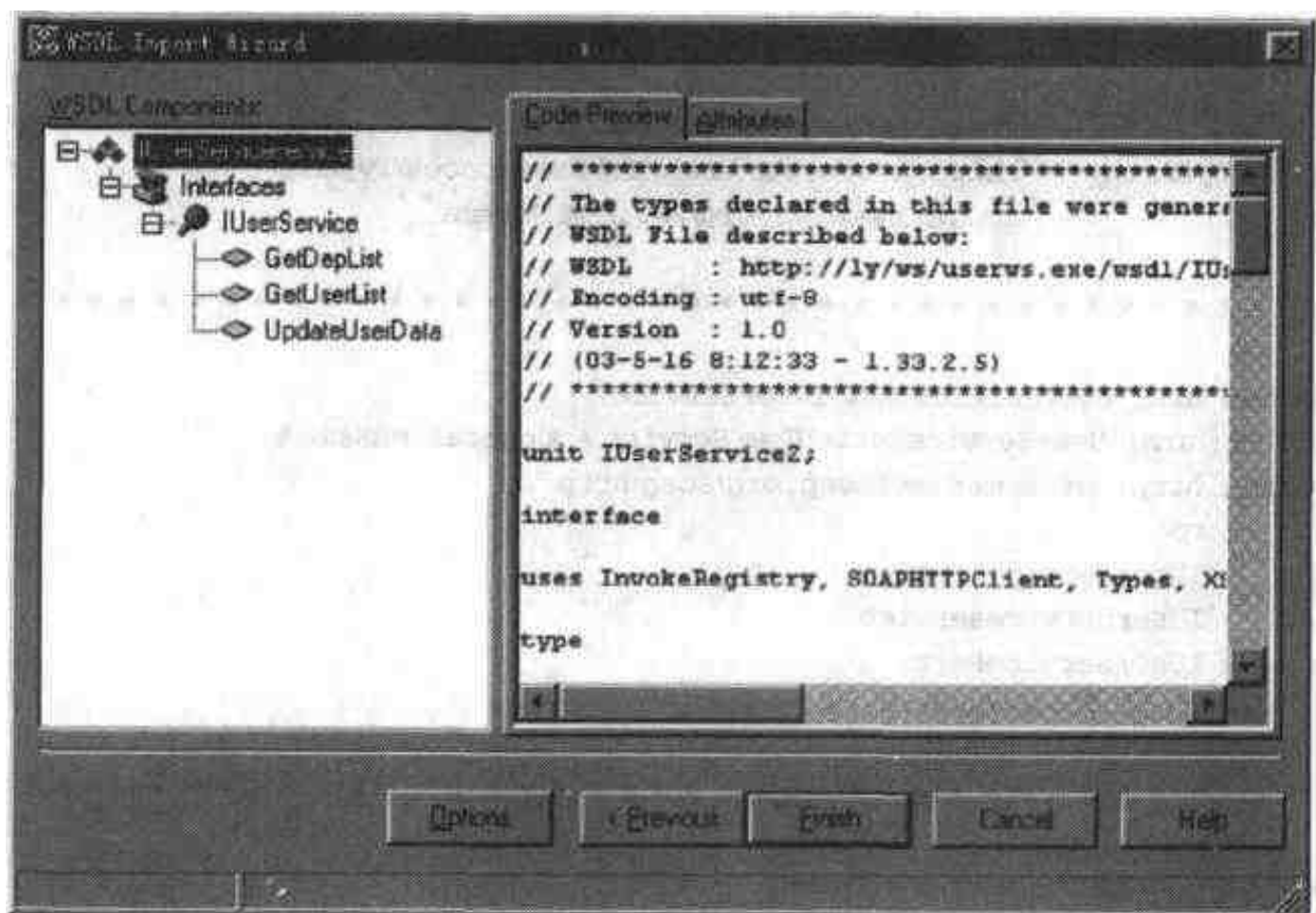


图 8-32 根据 Web Service 的 WSDL 自动创建接口定义文件

示例程序 8-14 Web Service 接口定义文件源代码

```

( * * * * * )
//
// The types declared in this file were generated from data read from the
// WSDL file described below:
// WSDL      : http://ly/ws/userws.exe/wsdl/IUserService
// Encoding  : utf-8
// Version   : 1.0
// (03-b-16 8: 20: 12 -1.33.2.5)
//
( * * * * * )
unit uIUserService;

interface

uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;

type

( * * * * * )
;

The following types, referred to in the WSDL document are not being represented
in this file. They are either aliases [ @ ] of other types represented or were
referred to but never [!] declared in the document. The types from the latter
category typically map to predefined/known XML or Borland types; however, they
could also indicate incorrect WSDL documents that failed to declare or import
a schema type.
;

( * * * * * )
//
// !: string      - "http://www.w3.org/2001/XMLSchema"
// !: TStringDynArray - "http://www.borland.com/namespaces/Types"
// !: int         - "http://www.w3.org/2001/XMLSchema"
//
( * * * * * )
//
// Namespace : urn: UserServiceIntf-IUserService
// soapAction: urn: UserServiceIntf-IUserService# %operationName%
// transport  : http://schemas.xmlsoap.org/soap/http
// style      : rpc
// binding    : IUserServicebinding
// service    : IUserServiceservice
// port       : IUserServicePort
// URL        : http://ly/ws/userws.exe/soap/IUserService
//
( * * * * * )

IUserService = interface (IInvokable)
['{949C2188-EA15-7F7F-ECC1-C47B2AA2DA3C}']
function GetDepList (out iCount: Integer): TStringDynArray; stdcall;
function GetUserList (const strName: WideString): WideString; stdcall;
function UpdateUserData (const UserData: WideString): Integer; stdcall;

```

```
end;
```

```
function GetIUserService (UseWSDL: Boolean = System.False; Addr: string = "";
HTTPIO: THTTPIO = nil): IUserService;
```

```
implementation
```

```
function GetIUserService (UseWSDL: Boolean; Addr: string; HTTPIO: THTTPIO):
IUserService;
```

```
const
```

```
  defWSDL = 'http://ly/ws/userws.exe/wsdl/IUserService';
```

```
  defURL = 'http://ly/ws/userws.exe/soap/IUserService';
```

```
  defSvc = 'IUserServiceservice';
```

```
  defPrt = 'IUserServicePort';
```

```
var
```

```
  RIO: THTTPIO;
```

```
begin
```

```
  Result := nil;
```

```
  if (Addr = "") then
```

```
  begin
```

```
    if UseWSDL then
```

```
      Addr := defWSDL
```

```
    else
```

```
      Addr := defURL;
```

```
  end;
```

```
  if HTTPIO = nil then
```

```
    RIO := THTTPIO.Create (nil)
```

```
  else
```

```
    RIO := HTTPIO;
```

```
  try
```

```
    Result := (RIO as IUserService);
```

```
    if UseWSDL then
```

```
    begin
```

```
      RIO.WSDLLocation := Addr;
```

```
      RIO.Service := defSvc;
```

```
      RIO.Port := defPrt;
```

```
    end else
```

```
      RIO.URL := Addr;
```

```
  finally
```

```
    if (Result = nil) and (HTTPIO = nil) then
```

```
      RIO.Free;
```

```
  end;
```

```
end;
```

```
initialization
```

```
  InvRegistry.RegisterInterface (TypeInfo (IUserService),
```

```
  'urn: UserServiceIntf-IUserService', 'utf-8');
```

```
  InvRegistry.RegisterDefaultSOAPAction (TypeInfo (IUserService),
```

```
  'urn: UserServiceIntf-IUserService# %operationName%');
```

```
end.
```

利用 Web Service 接口定义文件, 我们重新设置客户端应用程序的 HTTPRIO1 组件, 如图 8-33 所示。客户端应用程序除此之外无需进行任何修改。编译并运行, 我们发现运行结果和原来一样。进一步把客户端应用程序放到任何联网的一台机器上, 出现找不到 Web 地址的错误。原来我的调试网络中服务器没有提供域名解析服务 (DNS), 改用 IP 地址 `http://163.1.12.50/ws/userws.exe/wsdl/IUserService` 重新设置 HTTPRIO1 组件属性后, 结果一切运行正常。

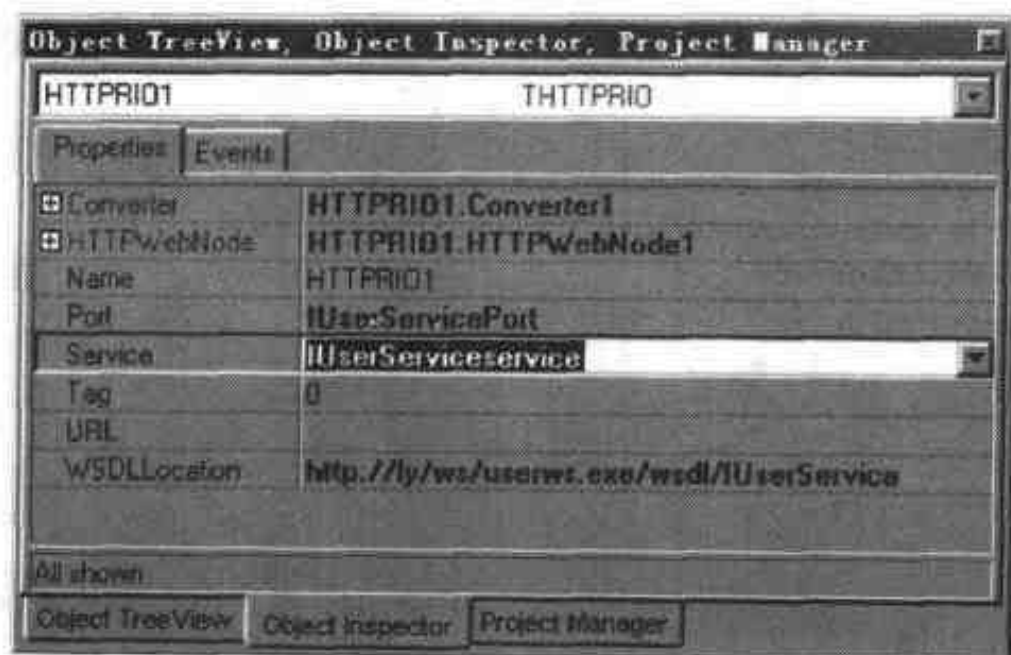


图 8-33 利用 Web Service 接口定义文件重新设置客户端应用程序的 HTTPRIO1 组件

回顾我们这个使用 Web Service 来封装业务对象的示例程序, 可以发现它的构架如图 8-34 所示。这里我们将接口、类分别放在独立的单元文件中, 每个单元只包含一个类。这样既为我们封装对象提供了灵活性, 同时也最大程度地实现了界面和业务的分离, 体现了一种良好的设计风格。

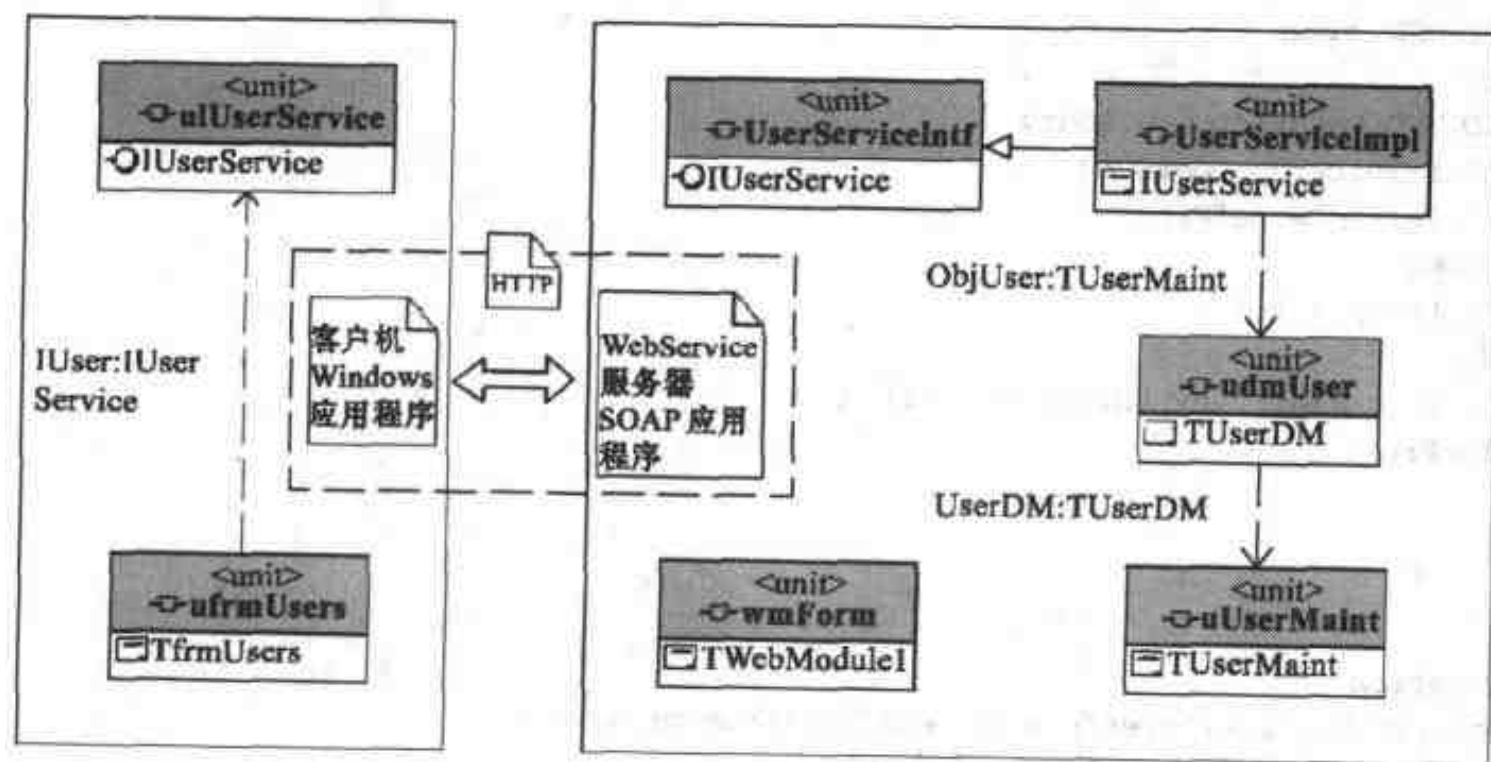


图 8-34 示例程序的类/单元关系及部署图

读者可以通过研习这个示例程序, 在编程中实现更优秀的面向对象编程思想。

Web 编程中的 Session 处理一直是一个难题。IntraWeb 提供了完整的 Session 功能，可以暂时或永久地保留一些重要的状态信息。同时它提供了 TSession 对象，通过面向对象的方式简化了使用 Session 的难度。

最重要的是利用 IntraWeb 开发的具有 Web Form 的 Web 应用程序，不需要在微软的 .NET 平台上运行，实际上即使是在 Windows 98 的 PWS (Personal Web Server) 上也可以运行。

可以说 IntraWeb 是 Delphi (C++ Builder) 特有的 Web Form 解决方案，是 Delphi 程序员进行 Web 开发的利器。

8.4.2 创建一个 Web Form 程序

为了让读者有一个感性认识，下面创建一个用户登录的 Web Form 程序，让我们体验一下 IntraWeb 的强大功能。该程序类似于示例程序 4-5 的 Windows 应用程序。

首先在 Delphi 的主菜单中选择 File|New|Other 菜单项，在弹出的 New Items 对话框中，找到 IntraWeb 选项页，选择 Stand Alone Application 图标，如图 8-36 所示。双击该图标，Delphi 将在指定目录下创建一个 IntraWeb 应用程序项目，如图 8-37 所示 (IWUnit2 单元是后加的)。

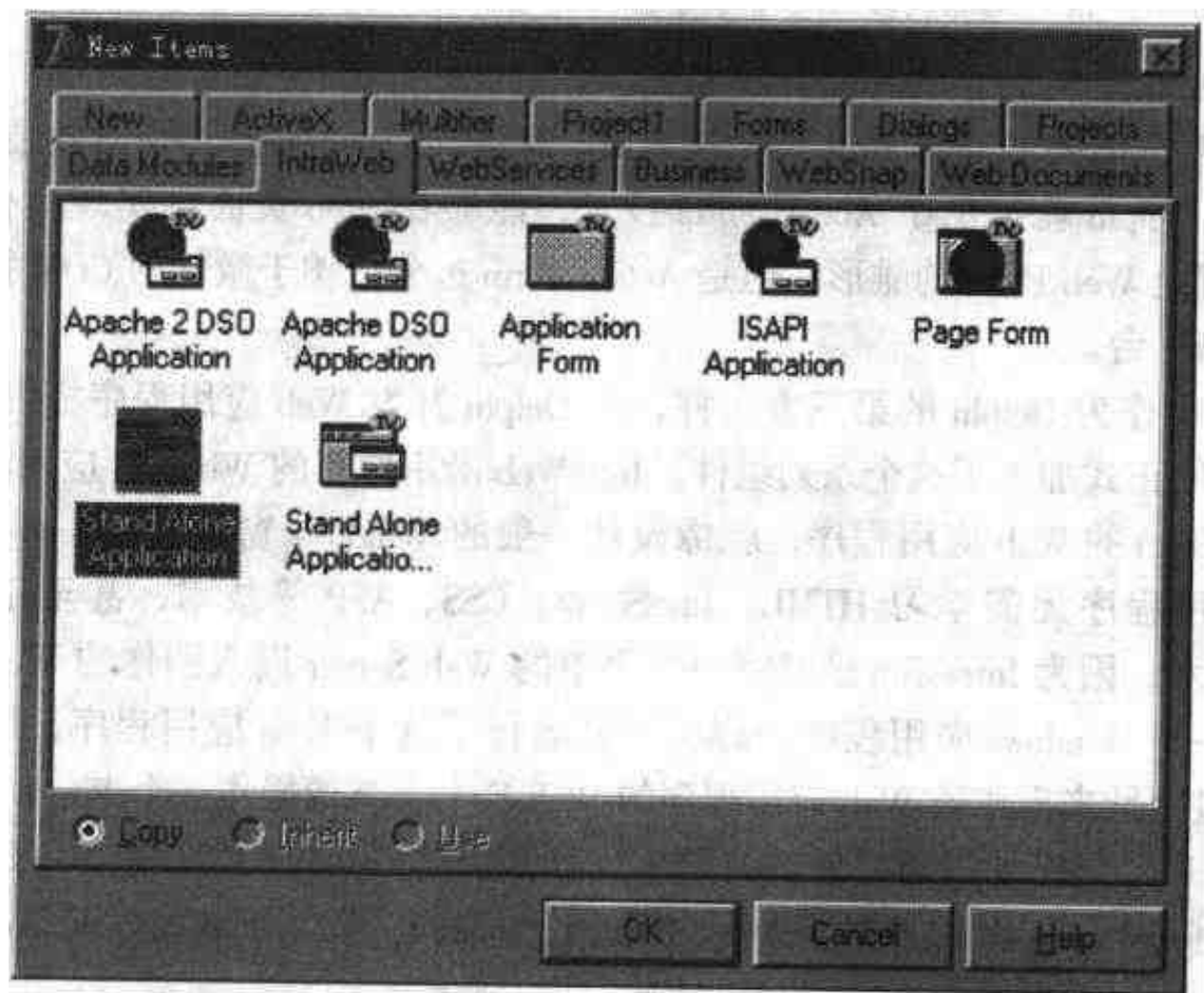


图 8-36 New Items 对话框的 IntraWeb 选项页

在图 8-36 中可以发现 IntraWeb 提供了多种类型的 Web 应用程序开发选项，其中：

- Apache 2 DSO Application 用于建立 Apache 2 DSO 的项目。
- Apache DSO Application 用于建立 Apache 1.x 的项目。
- Application Form 用于加入一个 Application Form 到现有项目中。
- Page Form 用于加入一个 Page Form 到现有项目中。

- Stand Alone Application 用于建立一个单独运行的内嵌 Web Server 的项目。
- Stand Alone Application With DataModule 用于建立一个单独运行的内嵌 Web Server 的项目，并在其中加入一个 DataModule，以支持数据库应用。

我们这里选用 Stand Alone Application 类型的项目，这是因为考虑到建立的应用程序包含了一个 Web Server，所以不需要另外安装 Web Server，也不需要特别的设置就可以在 Delphi 中进行调试和实际运行。

Delphi 为这个应用程序创建了以下主程序代码：

```
program IWProject;
{$PUBDIST}

uses
  IWInitStandAlone,
  ServerController in 'ServerController.pas'
  |IWServerController: TDataModule|,
  IWUnit1 in 'IWUnit1.pas' |formMain: TIWForm1|,
  IWUnit2 in 'IWUnit2.pas' |FormWelcome: TIWAppForm|;

{$R *.res}

begin
  IWRUN (TFormMain, TIWServerController);
end.
```

另外，在新建的 IntraWeb 应用程序项目中还包含了一个 ServerController 单元和一个 Application Form，其中的 ServerController 是控制核心，可以设置 Web Server 的诸如 Port、Application Name 等参数。

这个示例程序是用于网站登录的，我们从一个用户数据文件（login.dat）中取出用户的信息，以便在登录界面中对登录用户的合法性进行检查。在此将用户信息初始化部分程序写在 ServerController 中，其中用一个 TProfile 类来管理用户信息，用户信息对象将包含在对象集 ProfileList 中。

由于网页的特殊性，我们通常要用 Session 来保存一些持续性的状态变量。Session 在 IntraWeb 被处理成一个 TUserSession 对象。从 ServerController 单元中自动生成的关于 Session 的一段注释文字中我们知道，如果有需要保存信息的持续性状态变量，就应该写在 TUserSession 类中，作为它的数据成员调用，而不是写成一个全局变量。所以，我们将 TObjectList 类型的 ProfileList 变量以及存储当前登录用户信息的 TProfile 类型的 FMan 变量都写成 TUserSession 类的数据成员。这种以面向对象的思想来处理的 UserSession 对象，简化了 Session 的编程。

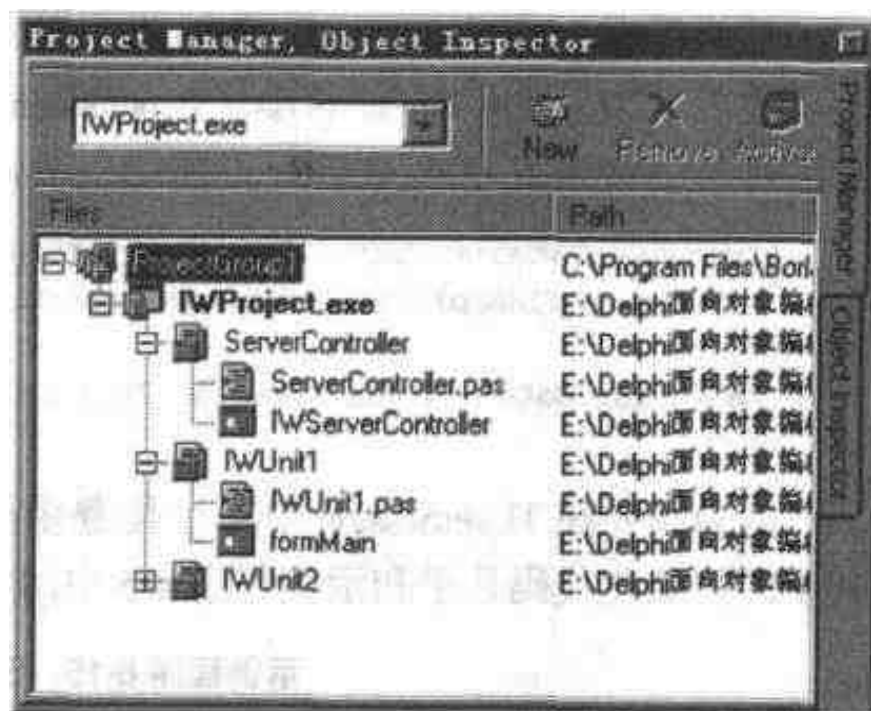


图 8-37 新创建一个 IntraWeb 应用程序项目

示例程序 8-15 是 ServerController 单元的源代码，注意此处将原来自动创建的 TUserSession = class 改成了 TUserSession = class (TComponent)，这样就将 TUserSession 对象的生命期交给了它的属主对象 TIWApplication 来管理。TUserSession 对象是这样创建的：

```
procedure TIWServerController.IWServerControllerBaseNewSession (
  ASession: TIWApplication; var VMainForm: TIWAppForm);
begin
  ASession.Data := TUserSession.Create (ASession);
end;
```

因为我要在 TUserSession 中初始化登录用户信息，所以覆盖了 TUserSession 的 Create 方法。其中的初始化代码几乎和示例程序 4-5 中的没有差别。

示例程序 8-15 ServerController 单元源代码

```
unit ServerController;
{PUBDIST}

interface

uses
  SysUtils, Classes, IWServerControllerBase,
  // For OnNewSession Event
  DBClient, Contnrs,
  IWApplication, IWAppForm;

type
  TIWServerController = class (TIWServerControllerBase)
    procedure IWServerControllerBaseNewSession (ASession: TIWApplication;
      var VMainForm: TIWAppForm);
  private
  public
  end;

  TProfile = class (TObject)
  public
    Name : String;
    Dep : String;
    Password : String;
    Job : String;
  end;

|
  This is a class which you can add variables to that are specific to the user. Add variables to this
  class instead of creating global variables. This object can references by using:
  UserSession
  So if a variable named UserName of type string is added, it can be referenced by using:
  UserSession.UserName
  Such variables are similar to globals in a normal application, however these variables are
  specific to each user.
  See the IntraWeb Manual for more details.
|
```

```

TUserSession = class (TComponent)
public
    ProfileList: TObjectList;
    ClientDataSet1: TClientDataSet;
    FMan: TProfile;
    constructor Create (AOwner: TComponent); override;
end;

// Procs
function UserSession: TUserSession;

implementation
{$R *.dfm}

uses
    IWInit;

{TUserSession}

constructor TUserSession.Create (AOwner: TComponent);
var
    i: integer;
    AMan: TProfile;
begin
    inherited;
    ProfileList: = TObjectList.Create (True);
    ClientDataSet1: = TClientDataSet.create (nil);
    FMan: = TProfile.Create;
    try
        ClientDataSet1.LoadFromFile (' login.dat ');
        ClientDataSet1.Active: = True;
        for i: = 1 to ClientDataSet1.RecordCount do
            begin
                AMan: = TProfile.Create;
                AMan.Name: = ClientDataSet1.FieldName (' Name') .AsString;
                AMan.Job: = ClientDataSet1.FieldName (' Job') .AsString;
                AMan.Dep: = ClientDataSet1.FieldName (' Dep') .AsString;
                AMan.Password: = ClientDataSet1.FieldName (' Password') .AsString;
                ProfileList.Add (AMan);
                ClientDataSet1.Next;
            end;
        finally
            ClientDataSet1.free;
        end;
    end;

function UserSession: TUserSession;
begin
    Result: = TUserSession (RWebApplication.Data);
end;

procedure TIWServerController.IWServerControllerBaseNewSession (
    ASession: TIWApplication; var VMainForm: TIWAppForm);
begin

```

```

    ASession.Data := TUserSession.Create (ASession);
end;

end.

```

接着，像设计一个普通的窗体一样，我们可以在 formMain 上放置 IW 组件，如图 8-38 用户登录界面所示。这里的下拉列表框 (TIWComboBox)、编辑框 (TIWEdit)、标签 (TIWLabel)、按钮 (TIWButton) 等组件的用法和以前使用的 VCL 控件几乎没有太大差别。此外，还使用了一个超链接组件 (TIWURL)，用于链接到新用户注册页面，就像我们在网上常见的那样。实际上，IntraWeb 所谓的 Application Form 就是一个 Web Form。

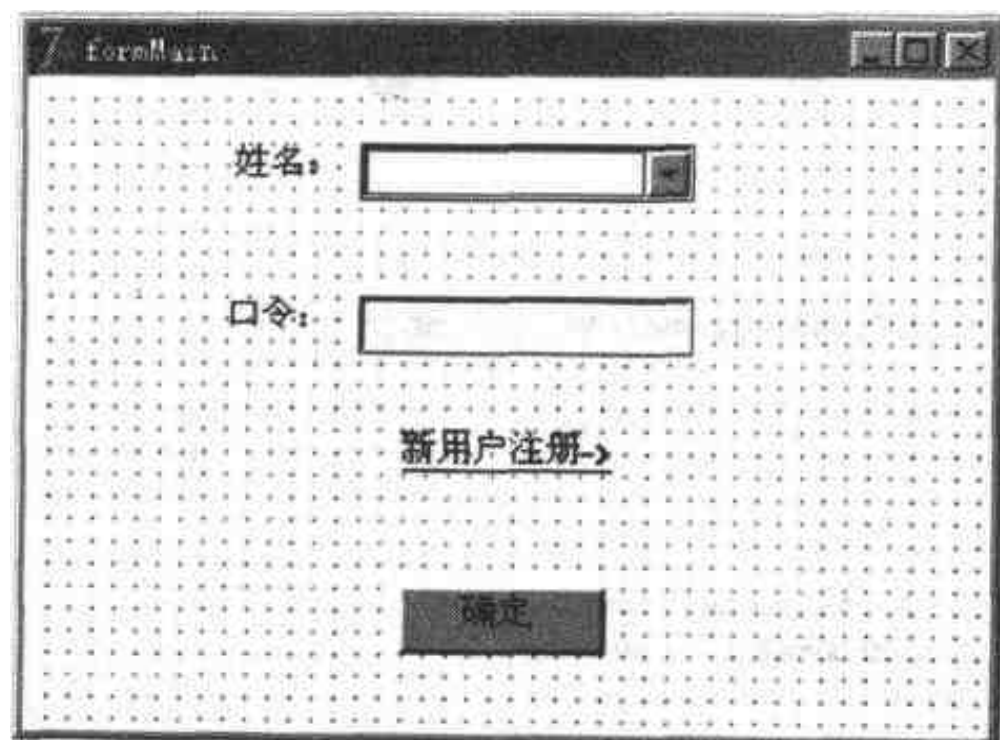


图 8-38 用户登录界面

示例程序 8-16 是用户登录界面 IWUnit1 单元的源代码。从中可见，在 Application Form 中编写代码感觉非常亲切，因为我们几乎可以不用学习其他的 Web 编程知识，而只要具备原来在 Delphi 中编程的经验即可。当然也有一些变化的地方要注意，比如在 Windows 程序中常写的对话框调用 Application.MessageBox，在这里就变成了 WebApplication.ShowMessage。在 IE 中弹出的对话框如图 8-39 所示。

示例程序 8-16 用户登录界面的源代码

```

unit IWUnit1;
{PUBDIST}

interface

uses
    IWAppForm, IWApplication, IWTypes, IWHTMLControls, IWCompLabel,
    IWCompListbox, IWCompEdit, Classes, Controls, IWControl,
    IWCompButton, SysUtils;

type
    TFormMain = class (TIWAppForm)
        IWButton1: TIWButton;
    end;

```

```

    IWEdit1: TIWEdit;
    IWComboBox1: TIWComboBox;
    IWLabel1: TIWLabel;
    IWLabel2: TIWLabel;
    iwuNewUser: TIWURL;
    procedure IWAppFormCreate (Sender: TObject);
    procedure IWButton1Click (Sender: TObject);
public
end;

implementation
{$R *.dfm}

uses
    ServerController, IWUnit2;

procedure TFormMain.IWAppFormCreate (Sender: TObject);
var
    i: integer;
    AMan: TProfile;
begin
    for i := 0 to UserSession.ProfileList.Count-1 do
    begin
        AMan := TProfile (UserSession.ProfileList.Items [i] );
        IWComboBox1.Items.Add (AMan.Name );
    end;
end;

procedure TFormMain.IWButton1Click (Sender: TObject);
var
    i: integer;
    Err: Boolean;
    AMan: TProfile;
begin
    Err := True;
    for i := 0 to (UserSession.ProfileList.Count-1) do
    begin
        AMan := TProfile (UserSession.ProfileList.Items [i] );
        if (trim (AMan.Name) = trim (IWComboBox1.Text)) and
            (trim (AMan.Password) = trim (IWEdit1.Text)) then
        begin
            Err := False;
            UserSession.FMan := AMan;
        end;
    end;
    if Err then
        WebApplication.ShowMessage ('非法用户,登录失败!', smAlert)
    else
    begin
        TFormWelcome.Create (WebApplication) .Show;
        Release;
    end;
end;

end.

```


那么在 Web 应用程序中能不能像 Windows 应用程序那样创建和使用多个 Web Form 呢？完全可以。下面将演示如何创建和使用多个 Web Form。

在主菜单中选择 File|New|Other 菜单项，在弹出的 New Items 对话框中，找到 IntraWeb 选项页，选择 Application Form 图标，如图 8-40 所示。双击该图标新建一个 Web Form。



图 8-39 在 IE 中弹出的对话框

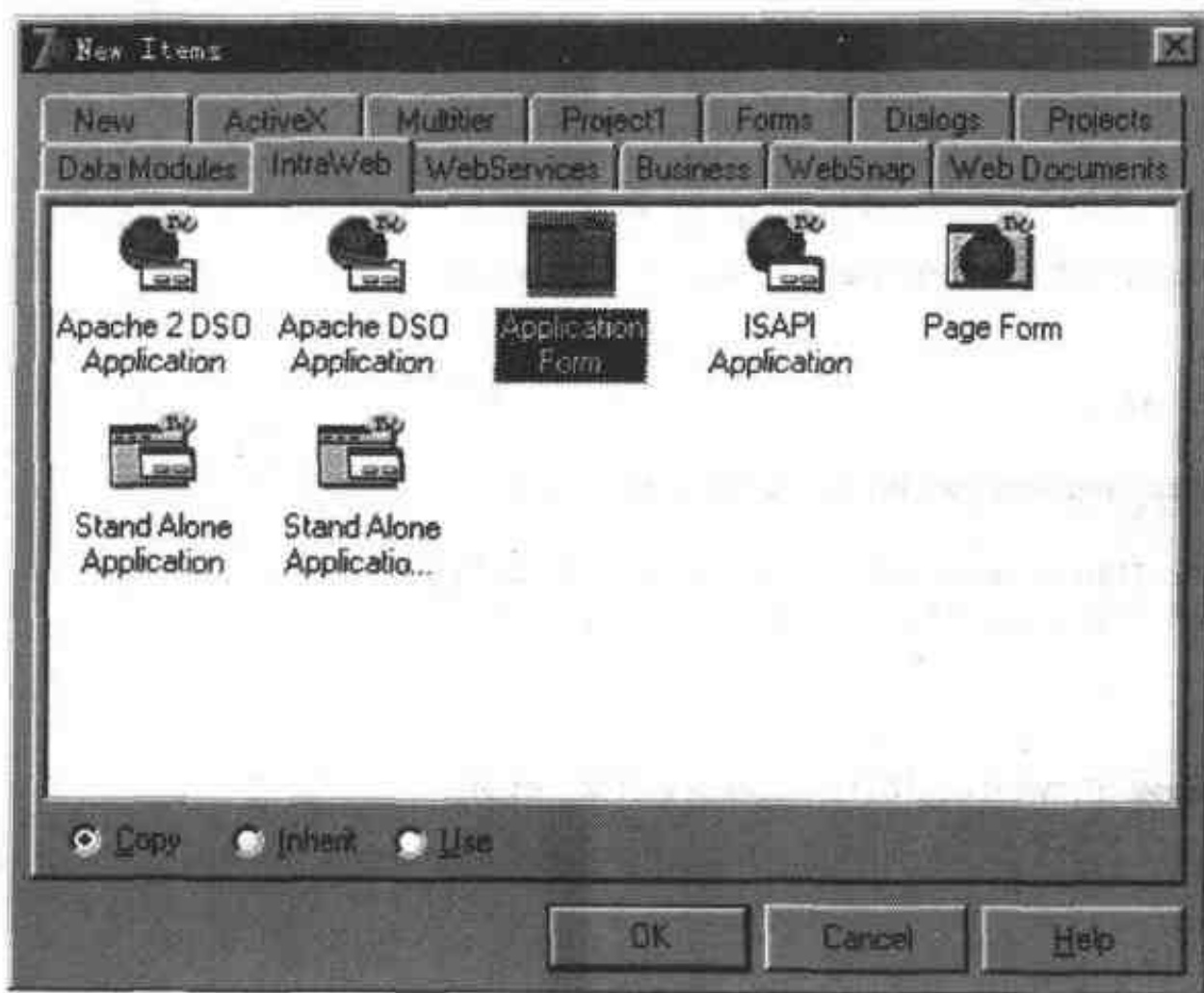


图 8-40 双击 IntraWeb 选项页的 Application Form 图标

将新建的单元命名为 IWUnit2，Application Form 命名为 FormWelcome，在其上拖放一个图形组件（TIWImage）、列表框组件（TIWListbox）和超链接组件（TIWURL），如图 8-41 所示。其中列表框中将显示当前登录用户的信息，该信息是利用 Session 技术通过 UserSession.FMan 保存和读取的。该单元的源代码如示例程序 8-17 所示。

为了调用和显示该 Web Form，我们在示例程序 8-16 中写进这样的代码：

```
TFormWelcome.Create(WebApplication).Show;  
Release;
```

是不是很熟悉？这里已经取消了 Form 的全局变量，而不再有这样的代码：

```
Var  
    FormWelcome: TFormWelcome
```

因为作为 Web 程序，Web Form 显示为一个网页，持续性信息通过 Session 保存，所以全局变量实在没有什么意义和必要。



图 8-41 FormWelcome 设计界面

示例程序 8-17 IWUnit2 的源代码

```

unit IWUnit2;
{$PUBDIST}

interface

uses

  IWAppForm, IWApplication, IWTypes, IWHTMLControls, IWCompLabel, jpeg,
  Classes, Controls, IWControl, IWExtCtrls, IWCompListbox;

type
  TFormWelcome = class (TIWAppForm)
    IWImage1: TIWImage;
    IWLabel1: TIWLabel;
    IWURL1: TIWURL;
    IWListbox1: TIWListbox;
    procedure IWAppFormCreate (Sender: TObject);
  public
  end;

implementation
{$R *.dfm}

uses
  ServerController;

procedure TFormWelcome.IWAppFormCreate (Sender: TObject);
begin
  if UserSession.FMan = nil then exit;
  IWListbox1.Items.Add (UserSession.FMan.Name);
  IWListbox1.Items.Add (UserSession.FMan.Dep);
end;

```

```
IWListBox1.Items.Add (UserSession.FMan.Job);  
end;  
  
end.
```

编译并运行该 Web 应用程序，这时自动启动一个内部的 Web Server。如图 8-42 所示，点击工具条上的第一个按钮，直接启动浏览器运行 Web Form。在 IE 中运行的登录界面如图 8-43 所示，登录成功后显示如图 8-44 所示的界面。

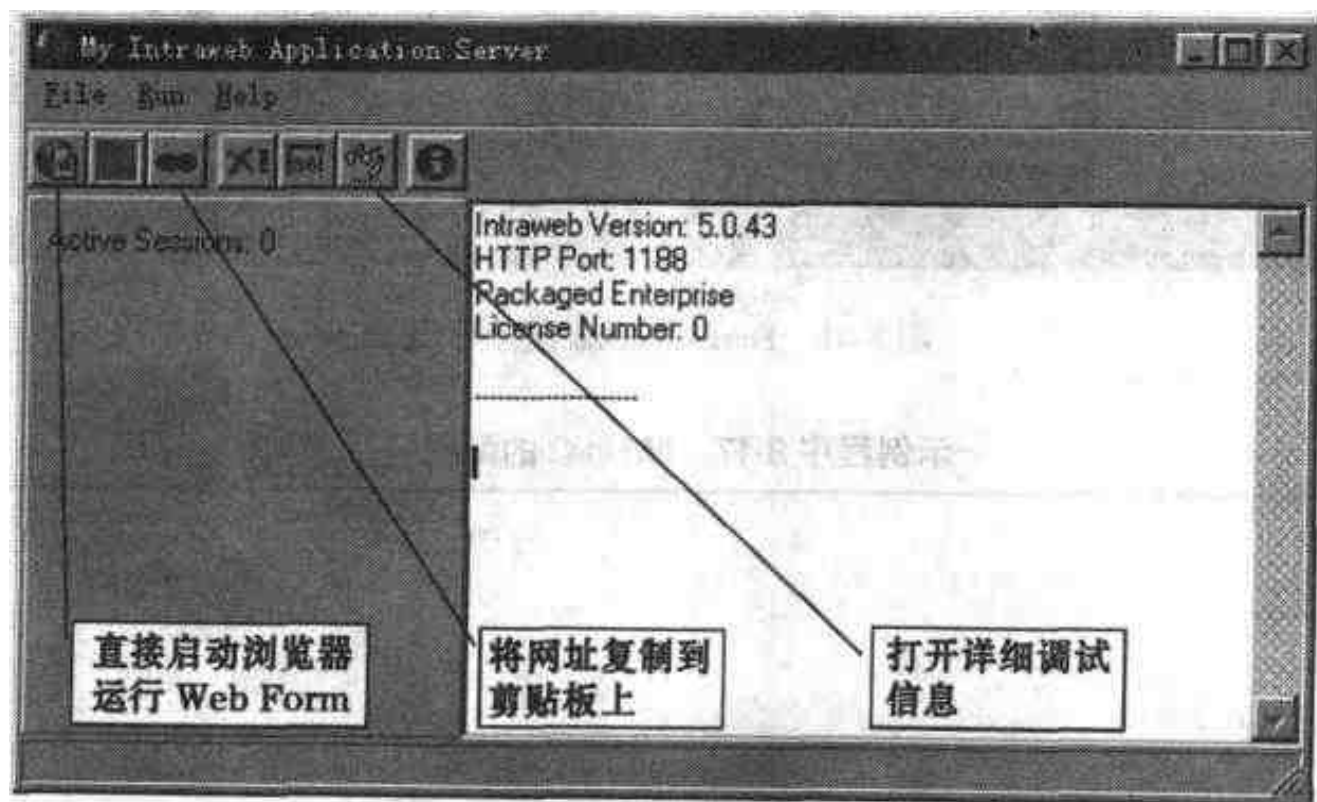


图 8-42 启动一个内部的 Web Server

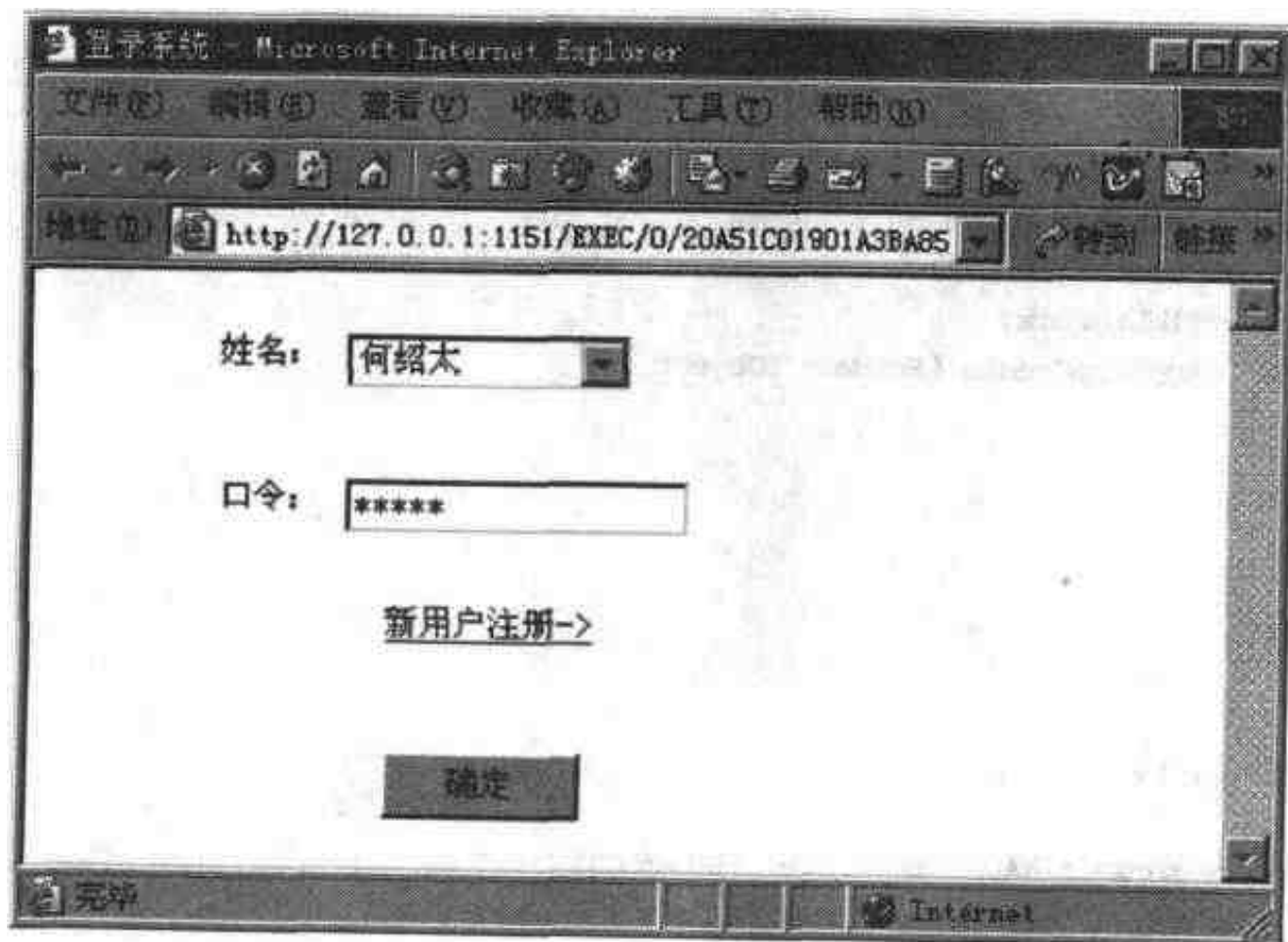


图 8-43 IE 中的登录界面

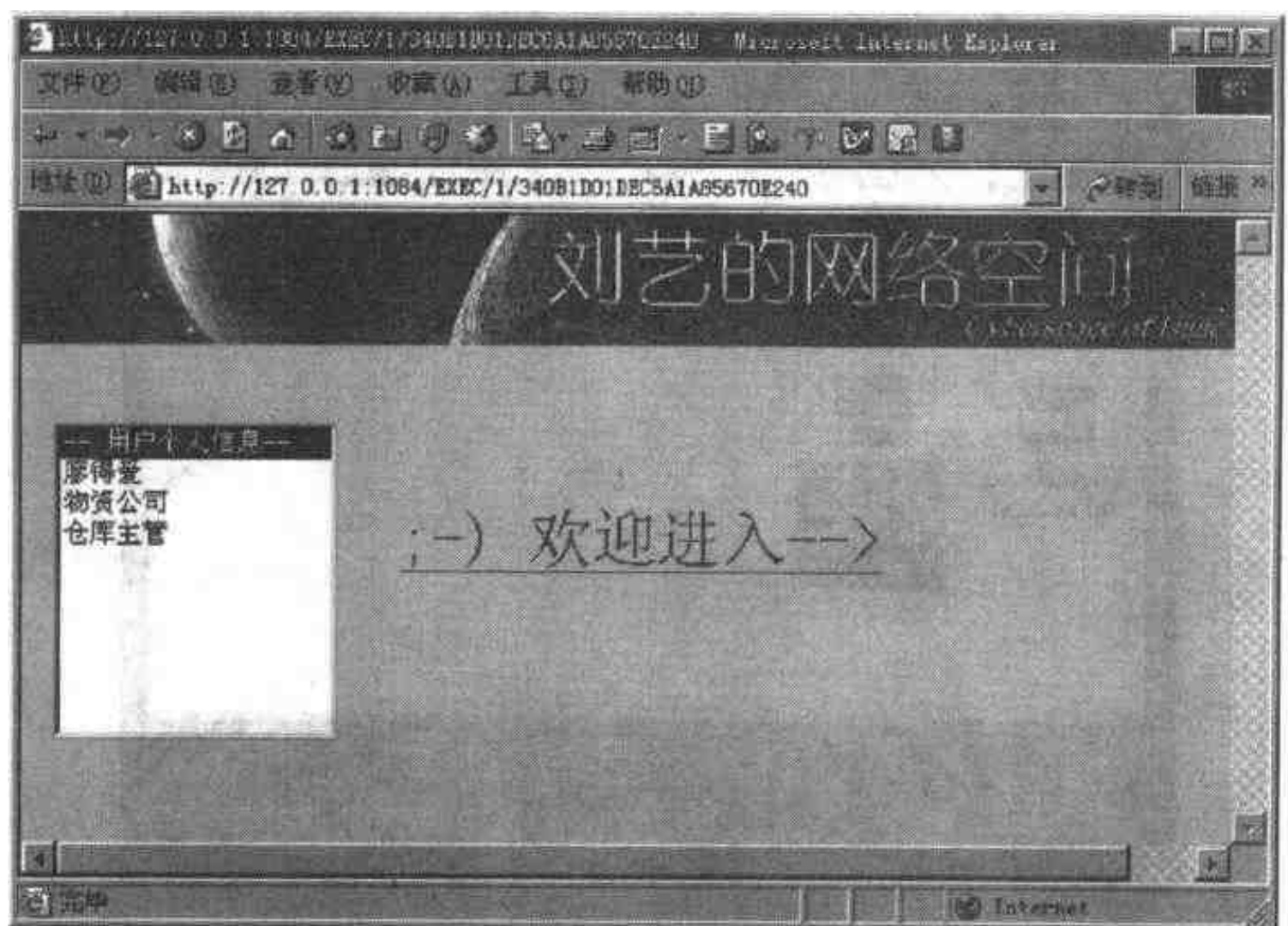


图 8-44 IE 中登录成功后的界面

8.4.3 IntraWeb 和业务对象整合

以 Web Form 作为客户端的操作界面有跨平台的优势,所以在使用 Web Form 时,我们应该充分发挥其在界面上的特长,而不应在其中包含业务内容。这就是说,Web Form 也应该体现“瘦”的原则,体现界面和业务分离的原则。

下面通过示例程序来讨论一下如何将 IntraWeb 和业务对象整合?如何将一个 Windows 应用程序的界面改造成 Web Form 界面,并使用原来的业务对象?这是一个十分有意义的话题,因为将一个 Windows 应用迁移到 Web 应用,如果没有一个简便易行的方法,将会带来巨大的风险和开销。

下面将举一个和数据库应用有关的例子,它来自前面的“界面和业务分离的演化实例”(8.2 节)。为支持数据库应用开发,IntraWeb 提供了 Stand Alone Application With DataModule 向导。在 Delphi 的主菜单中选择 File|New|Other 菜单项,在弹出的 New Items 对话框中,找到 IntraWeb 选项页,双击 Stand Alone Application With DataModule 图标,如图 8-45 所示

此时 Delphi 创建了一个带有数据模块的 IntraWeb 应用程序项目,我们将示例程序 8-3 加入到项目中,并将 IntraWeb 数据模块单元命名为 udmUser。然后我们将示例程序 8-4 数据模块中的组件复制到新建的 IntraWeb 数据模块中,并将新数据模块重新命名为 UserDM。最后在项目中将包含如图 8-46 所示的那些文件。

在 IWUnit1 单元的 formMain 上拖放 IW 组件和数据库相关组件,如图 8-47 所示。读者可以

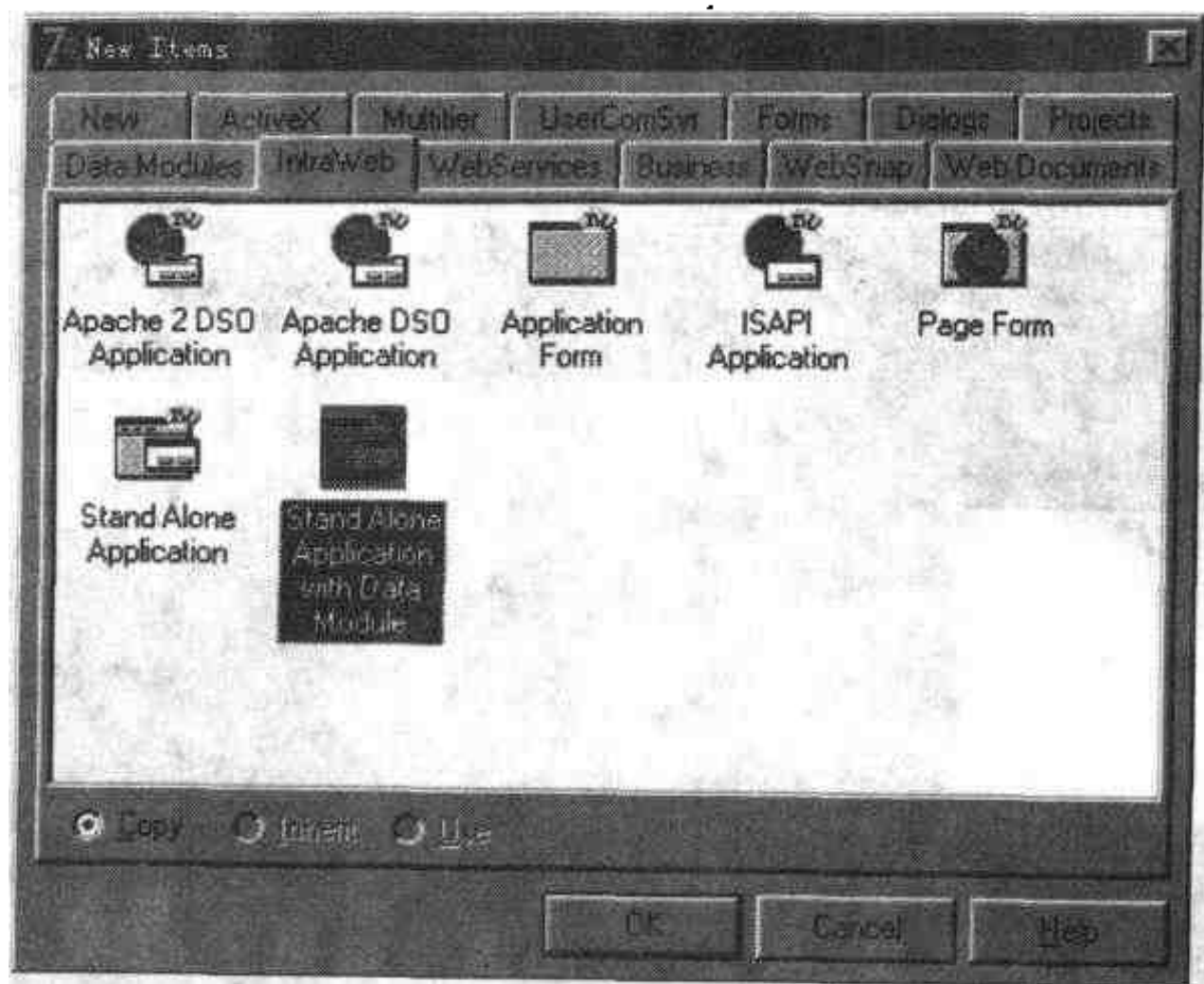


图 8-45 在 IntraWeb 选项页，双击 Stand Alone Application With DataModule 图标

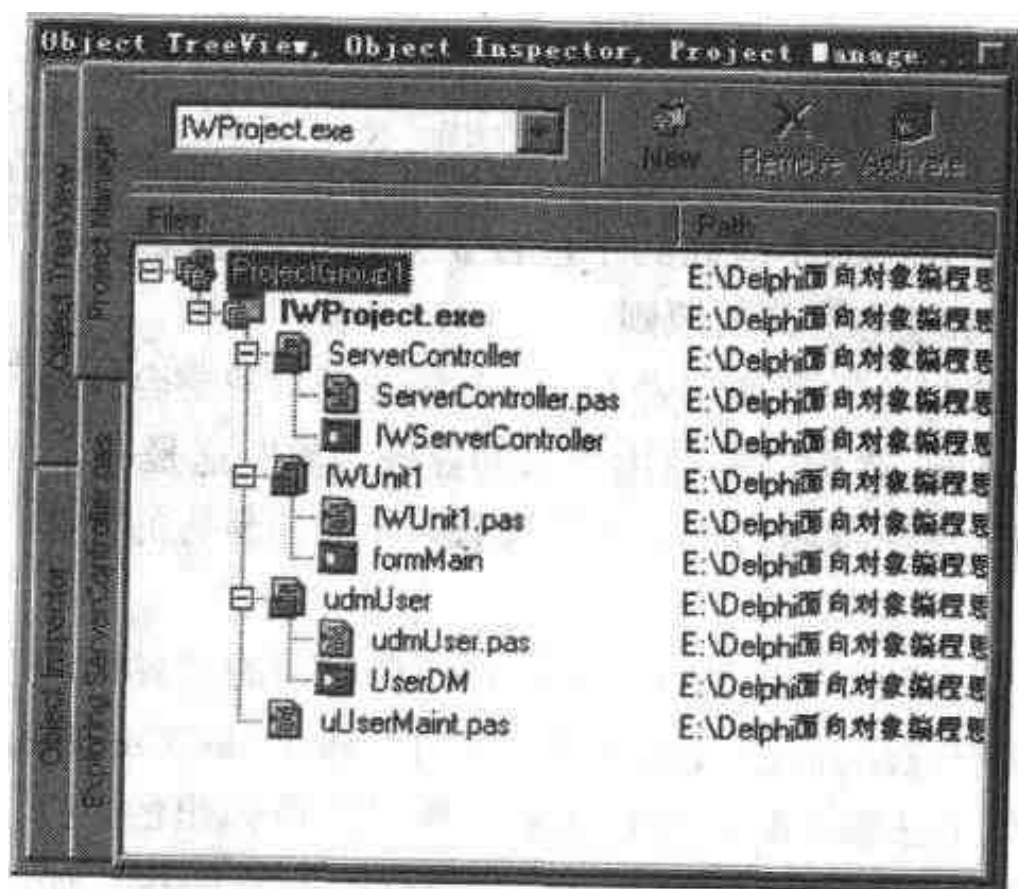


图 8-46 含有数据模块的 IntraWeb 应用程序项目

发现该界面最终可以设计成图 8-5 的样子。界面单元的源代码如示例程序 8-18 所示，大多数代码可以直接从示例程序 8-2 中复制过来，而只需进行一些简单的修改即可。

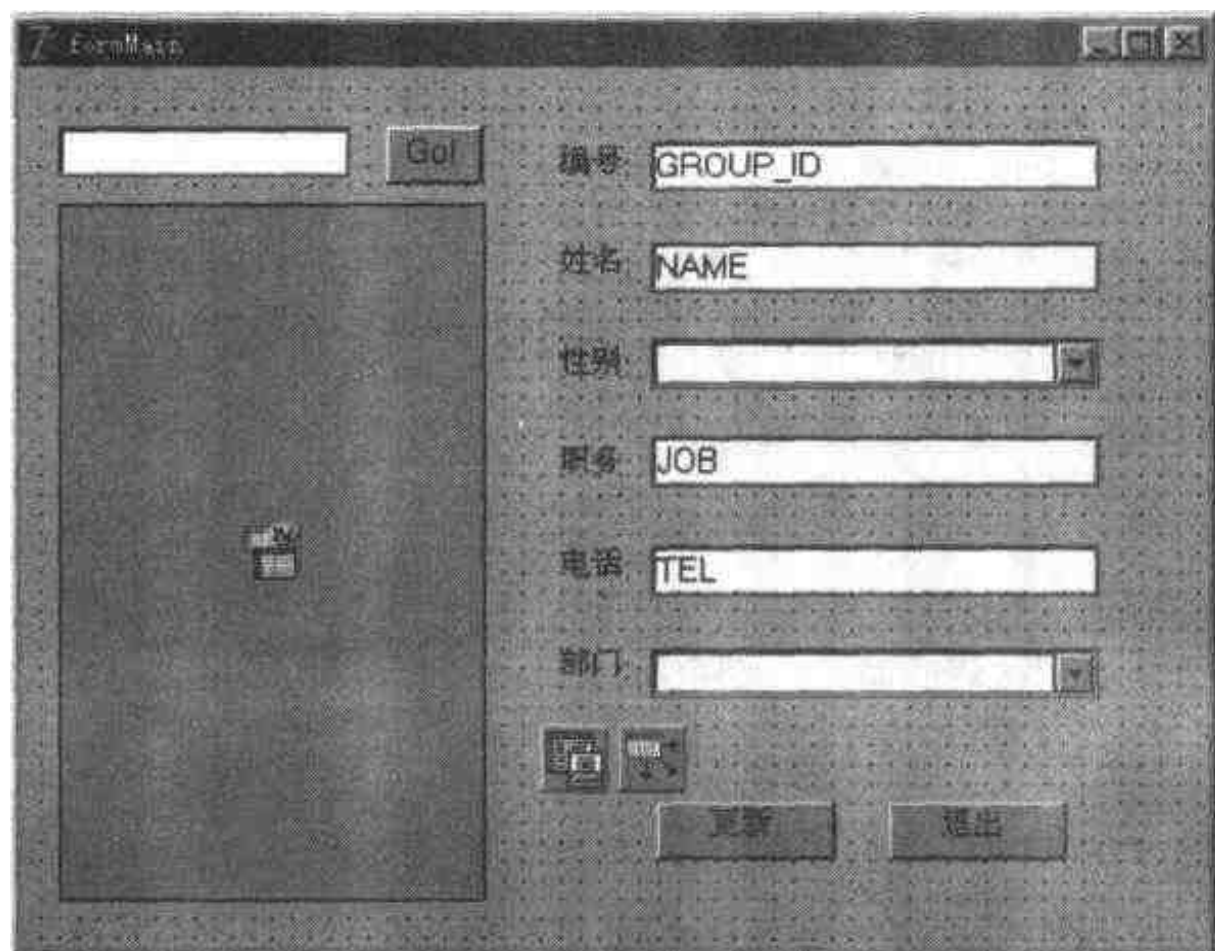


图 8-47 Web Form 界面的设计

示例程序 8-18 Web Form 界面单元源代码

```

unit IWUnit1;
|PUBDIST|

interface

uses
  IWAppForm, IWApplication, IWTypes, IWCompLabel, DB, DBClient,
  IWCompListbox, IWDBStdCtrls, IWCompButton, IWCompEdit, Classes, Controls,
  IWControl, IWGrids, Variants, IWDBGrids;

type
  TFormMain = class (TIWAppForm)
    IWDBGrid1: TIWDBGrid;
    edtQryByName: TIWEdit;
    btnQryByName: TIWButton;
    IWDBEdit1: TIWDBEdit;
    IWDBEdit2: TIWDBEdit;
    IWDBComboBox1: TIWDBComboBox;
    IWDBEdit3: TIWDBEdit;
    IWDBEdit4: TIWDBEdit;
    dbcDep: TIWDBComboBox;
    btnUpdate: TIWButton;
    btnExit: TIWButton;
    DataSource1: TDataSource;
    IWLabel1: TIWLabel;
    IWLabel2: TIWLabel;
    IWLabel3: TIWLabel;
  end;

```

```

    IWLabel4: TIWLabel;
    IWLabel5: TIWLabel;
    IWLabel6: TIWLabel;
    cdsUserMaint: TClientDataSet;
    procedure btnQryByNameClick (Sender: TObject);
    procedure btnUpdateClick (Sender: TObject);
    procedure IWAppFormCreate (Sender: TObject);
    procedure IWDBGrid1Columns0Click (ASender: TObject;
        const AValue: String);
    procedure btnExitClick (Sender: TObject);
public
end;

implementation
{$R *.dfm}

uses
    ServerController, udmUser, uUserMaint;

procedure TFormMain.btnQryByNameClick (Sender: TObject);
var
    objUsers: TUserMaint;
begin
    objUsers := TUserMaint.create;
    cdsUserMaint.Data := objUsers.GetUserList (edtQryByName.Text);
    cdsUserMaint.Active := true;
    btnUpdate.Enabled := true;
    objUsers.Free;
end;

procedure TFormMain.btnUpdateClick (Sender: TObject);
var
    objUsers: TUserMaint;
    nErr: Integer;
begin
    objUsers := TUserMaint.create;
    try
        if cdsUserMaint.State = dsEdit then cdsUserMaint.Post;
        if (cdsUserMaint.ChangeCount > 0) then
            begin
                objUsers.UpdateUserData (cdsUserMaint.Delta, nErr);
                if nErr > 0 then
                    WebApplication.ShowMessage ('更新失败!', smAlert, '操作提示')
                else
                    begin
                        WebApplication.ShowMessage ('更新成功!', smAlert, '操作提示');
                        btnQryByNameClick (nil);
                    end;
            end;
    end;
finally
    objUsers.Free;
end;
end;

```



```
procedure TFormMain.IWAppFormCreate (Sender: TObject);
var
  objUsers: TUserMaint;
begin
  objUsers := TUserMaint.create;
  dbcbDep.Items.Assign (objUsers.GetDepList);
  btnUpdate.Enabled := true;
  objUsers.Free;
end;

procedure TFormMain.IWDBGrid1Columns0Click (ASender: TObject;
  const AValue: String);
begin
  DataSource1.DataSet.Locate ('ID', AValue, []);
end;

procedure TFormMain.btnExitClick (Sender: TObject);
begin
  WebApplication.Terminate ('感谢使用,再见!');
end;

end.
```

为什么此 Windows 应用程序的界面单元能轻松地改成 Web 应用程序的 Web Form? 除了 IntraWeb 方便好用外, 一个重要的原因就是我们在原来程序设计中遵循了界面和业务分离的原则。特别是业务对象的使用大大降低了界面中的代码数量和复杂程度。另外, 由于业务对象和 Web 界面无关, 因此原来的业务对象几乎不需改动。例如, 业务对象 TUserMaint 的单元 uUserMaint 就是直接使用原来的, 惟一的改动是删除了构造方法 create 和析构方法 destroy, 因为不再需要创建和管理数据模块对象 UserDM 了。UserDM 的创建和管理工作交给了 ServerController 单元的 TUserSession 对象, 如示例程序 8-20 所示。

这就是说, 将一个利用面向对象思想进行良好设计的 Windows 应用程序迁移到 Web 应用, 对于 IntraWeb 技术而言, 不仅简便易行, 而且降低了风险。其中, 最主要的工作仅在于界面的变化, 即由 Windows Form 变成了 Web Form, 而业务却保持了相对稳定。

另外, 我们从中还可以悟出这样一个道理, 好的开发工具要求程序员有好的编程思想而不是有更多的编程技巧。我们用 Delphi 开发这样一个基于数据库应用的 Web 应用程序, 实际上并没有用到太多的编程技巧。

从图 8-48 中可以发现, IntraWeb 数据模块中使用的数据库组件和原来 Windows 程序中的一样。示例程序 8-19 所示的数据模块单元的源代码改动也极小。

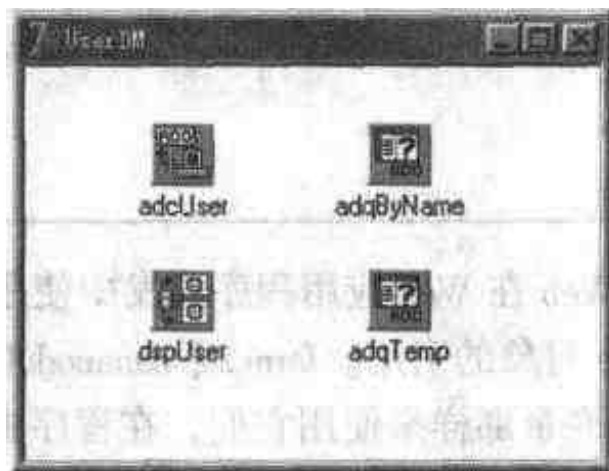


图 8-48 IntraWeb 数据模块中使用的数据库组件和原来 Windows 程序中的一样

示例程序 8-19 数据模块单元的源代码

```

unit udmUser;

interface

uses
  {$IFDEF Linux} QForms, {$ELSE} Forms, {$ENDIF}
  SysUtils, Classes, Provider, DB, ADODB;

type
  TUserDM = class (TDataModule)
    adcUser: TADOConnection;
    adqByName: TADOQuery;
    dspUser: TDataSetProvider;
    adqTemp: TADOQuery;
    procedure DataModuleCreate (Sender: TObject);
  private
  public
  end;

// Procs
function UserDM: TUserDM;

const
  ADO__STRING = ' Provider = Microsoft.Jet.OLEDB.4.0; Data Source = wz.mdb; Persist Security Info = False';

implementation
{$R *.dfm}
uses
  IWinInit,
  ServerController;

function UserDM: TUserDM;
begin
  Result := TUserSession (RWebApplication.Data) .UserDM;
end;

procedure TUserDM.DataModuleCreate (Sender: TObject);
begin
  adcUser.ConnectionString := ADO__STRING;
end;

end.

```

IntraWeb 在 Web 应用程序开发中使用了线程技术，所以我们不能用全局变量来存储 form 或 datamodule 对象的引用。form 或 datamodule 通常是存储在 WebApplication.Data 中的，为了能够像使用全局变量那样来使用它们，在程序中使用这样的函数：

```

function UserDM: TUserDM;
begin
  Result := TUserSession (RWebApplication.Data) .UserDM;
end;

```

这样一来, 在使用 UserDM 时, 并不是使用一个全局变量, 而是使用了一个返回值类型是 TUserDM 的函数。当然, 知道了其中的道理也不一定非要使用这样的函数。我们也可以直接使用下面的方法来调用:

```
TDataModule1 (WebApplication.Data);
```

查看示例程序 8-20 所示的 ServerController 单元的源代码, 我们看到数据模块对象是作为 TUserSession 对象的数据成员创建的, 所以它可以持续保存在 Session 中。

示例程序 8-20 ServerController 单元的源代码

```
unit ServerController;
{PUBDIST}

interface

uses
  Classes,
  udmUser,
  IWServerControllerBase, IWAppForm, IWApplication,
  SysUtils;

type
  TIWServerController = class (TIWServerControllerBase)
    procedure IWServerControllerBaseNewSession (ASession: TIWApplication;
      var VMainForm: TIWAppForm);
  private
  public
  end;

  TUserSession = class (TComponent)
  public
    UserDM: TUserDM;
    constructor Create (AOwner: TComponent); override;
  end;

// Procs
function UserSession: TUserSession;

implementation
{$R *.dfm}

uses
  IWInit;

function UserSession: TUserSession;
begin
  Result := TUserSession (RWebApplication.Data);
end;

{TUserSession}

constructor TUserSession.Create (AOwner: TComponent);
begin
  inherited;
```

```

    UserDM: = TUserDM.Create (AOwner);
end;

procedure TIWServerController.IWServerControllerBaseNewSession (
    ASession: TIWApplication; var VMainForm: TIWAppForm);
begin
    ASession.Data := TUserSession.Create (ASession);
end;

end.

```

调试并运行该程序，出现了如图 8-49 所示的异常，数据库无法使用。不少使用过 IntraWeb 开发数据库的朋友都遇到过这个问题，但发现使用 DBE 没有这个问题，于是网上有人武断地下结论说 IntraWeb 不支持微软的 ADO。但情况并非如此！



图 8-49 使用 ADO 组件时出现“尚未调用 CoInitialize”错误

实际上这个问题的本质是调用 COM 的问题，而不是 ADO 的问题！在 IntraWeb 中调用 COM 对象，都需要将 ServerController 中的 ComInitization 设为 ciNormal 或是 ciMultiThreaded，如图 8-50 所示。

注意 在使用 Delphi 的 WebSnap 技术开发 Web 应用程序时，也存在类似的问题。

最后运行程序，IE 中成功出现如图 8-51 所示的运行结果。这个 Web Form 的外观和操作都类似于原来 Windows 程序的 Form。这就是说，程序的用户界面通过 Web 浏览器可以延伸到任何能上网的终端设备上，实现了跨平台应用。

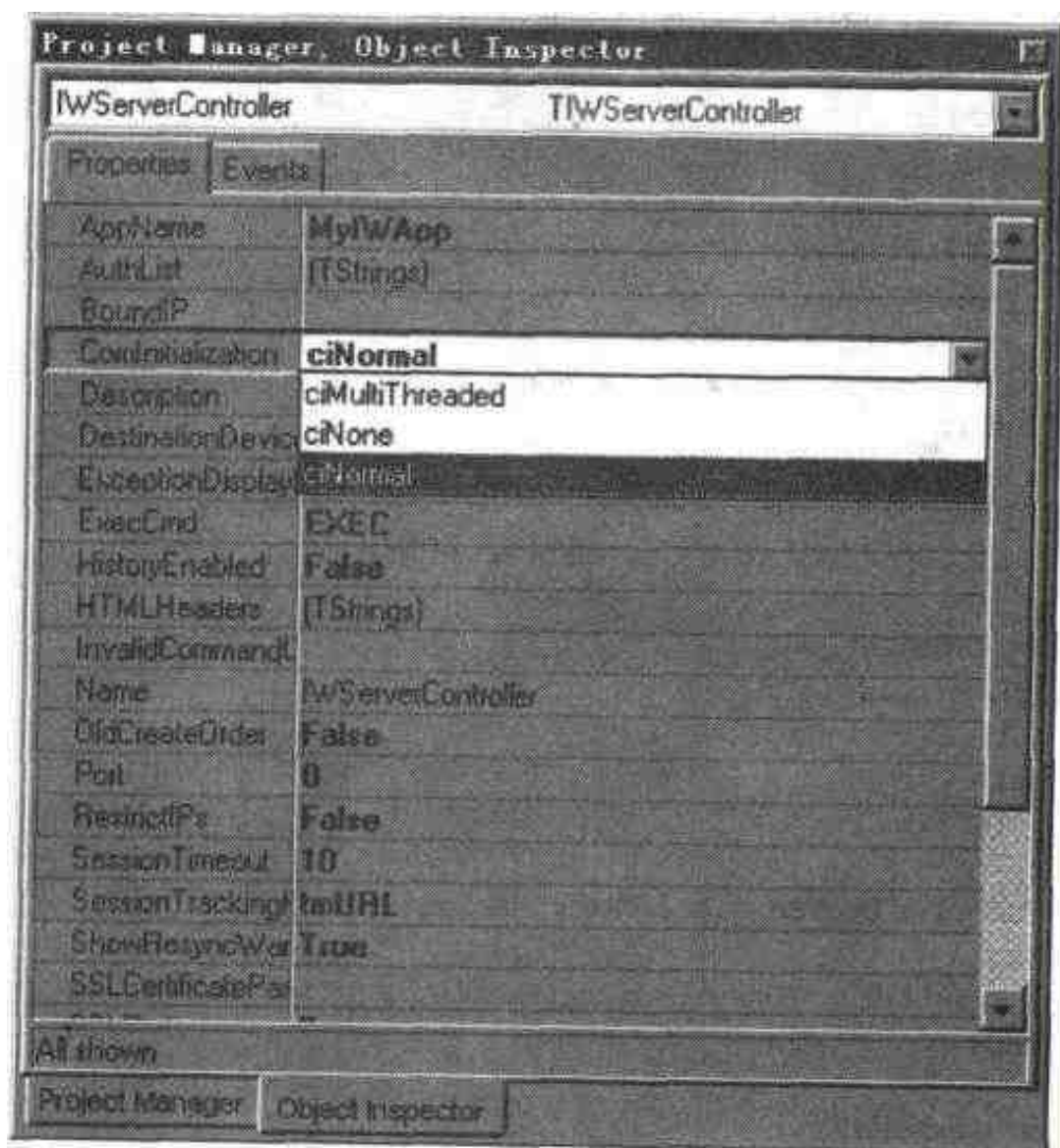


图 8-50 使用 COM 对象需要设置 ServerController 中的 ComInitization 属性

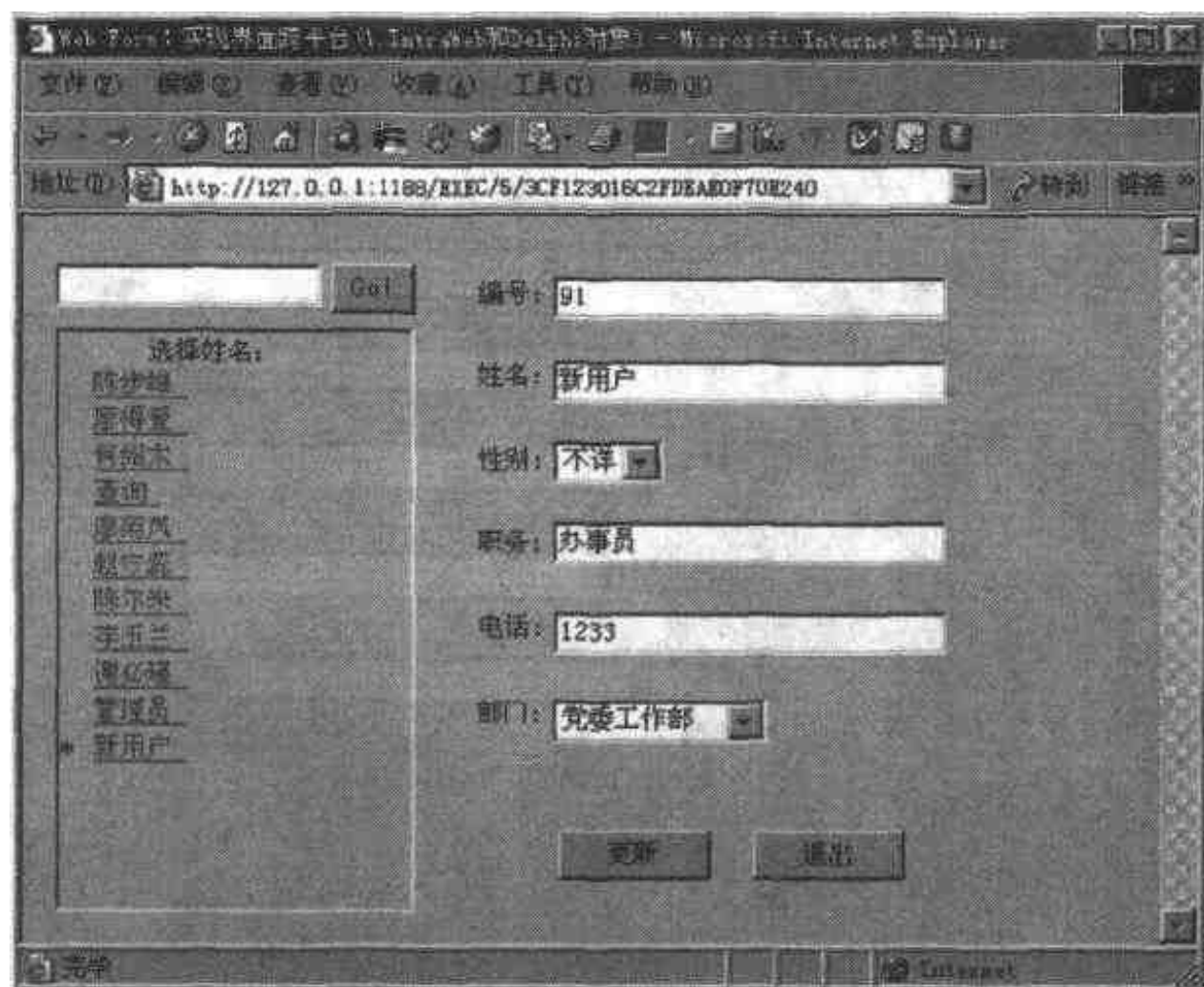


图 8-51 IE 中的运行结果和 Windows 程序的 Form 相似

8.4.4 IntraWeb 和 Web Service 整合

如果界面跨平台能结合业务跨平台，将不同平台上的业务封装和连接起来，并整合到统一的 Web Form 界面上，实现最佳的分布式应用解决方案该有多好！下面我将通过 IntraWeb 和 Web Service 的整合，演示一下这种灵活的架构。最重要的是我们不需要新增其他工具，就可以使用 Delphi 实现这种架构，由此可见 Delphi 的强大功能。

8.3 节已经实现了一个封装了业务的 Web Service。我们现在创建一个 IntraWeb 的 Stand Alone Application 项目，在项目中加入 Web Service 的接口文件 uUserService.pas，该文件如示例程序 8-14 所示。

注意 IntraWeb 可以和任何 Web Service 整合，不一定是 Delphi 开发的 Web Service。对于其他 Web Service 一样可以利用如图 8-30 所示的 WSDL 向导导入接口文件。

接着把项目中的 IWUnit1 换成示例程序 8-18，并加上一个用于调用 Web Service 的 THTTPIO 组件，如图 8-52 所示。另外，为了演示 Web Form 界面在浏览器中更换背景的效果，我在界面上还放置了一个 TIWCheckBox 组件 IWCheckBox1。

最后，项目文件组成如图 8-53 所示。这个 IntraWeb 的 Stand Alone Application 程序，仅仅是用来绑定业务，实现一个界面。它符合“瘦”的标准。

接下来修改 IWUnit1 中的源代码，使之如示例程序 8-21 所示。这段绑定 Web Service 的 Web Form 界面程序类似于示例程序 8-13。

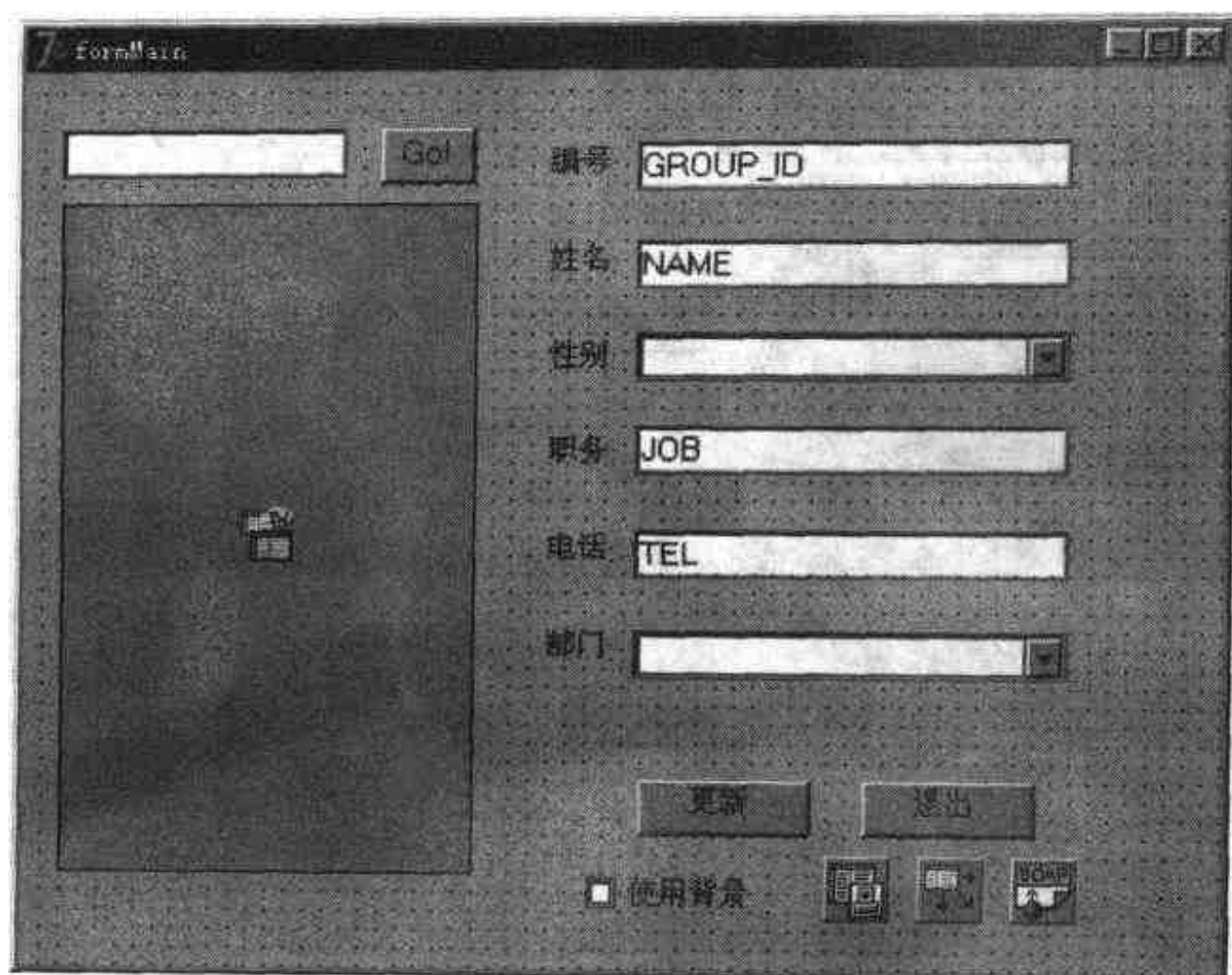


图 8-52 能调用 Web Service 的 Web Form

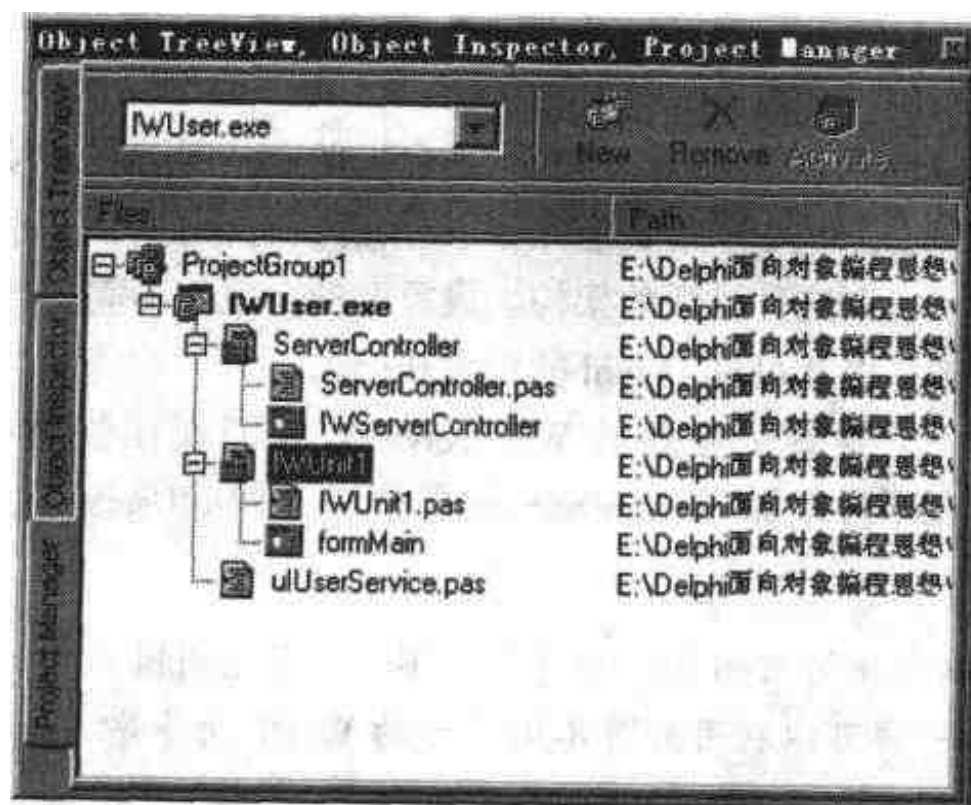


图 8-53 一个 IntraWeb 和 Web Service 整合的项目

示例程序 8-21 绑定 Web Service 的 Web Form 界面程序源代码

```
unit IWUnit1;
{PUBDIST}

interface
```

```

uses
  IWAppForm, IWApplication, IWTypes, IWCompLabel, DB, DBClient,
  IWCompListbox, IWDBStdCtrls, IWCompButton, IWCompEdit, Classes, Controls,
  IWControl, IWGrids, IWDBGrids,
  types, Variants, InvokeRegistry, Rio, SOAPHTTPClient, DBTables,
  IWClientSideDatasetBase, IWClientSideDatasetDBLink, IWDynGrid,
  IWCompCheckbox;

type
  TFormMain = class (TIWAppForm)
    IWDBGrid1: TIWDBGrid;
    edtQryByName: TIWEdit;
    btnQryByName: TIWButton;
    IWDBEdit1: TIWDBEdit;
    IWDBEdit2: TIWDBEdit;
    IWDBComboBox1: TIWDBComboBox;
    IWDBEdit3: TIWDBEdit;
    IWDBEdit4: TIWDBEdit;
    dbcbDep: TIWDBComboBox;
    btnUpdate: TIWButton;
    btnExit: TIWButton;
    DataSource1: TDataSource;
    IWLabel1: TIWLabel;
    IWLabel2: TIWLabel;
    IWLabel3: TIWLabel;
    IWLabel4: TIWLabel;
    IWLabel5: TIWLabel;
    IWLabel6: TIWLabel;
    HTTPRIO1: THHTTPRIO;
    cdsUserMaint: TClientDataSet;
    IWCheckBox1: TIWCheckBox;
    procedure btnQryByNameClick (Sender: TObject);
    procedure btnUpdateClick (Sender: TObject);
    procedure IWAppFormCreate (Sender: TObject);
    procedure IWDBGrid1Columns0Click (ASender: TObject;
      const AValue: String);
    procedure btnExitClick (Sender: TObject);
    procedure IWCheckBox1Click (Sender: TObject);
  end;

implementation
{$R *.dfm}

uses
  ServerController, uUserService;

procedure TFormMain.btnQryByNameClick (Sender: TObject);
var
  IUser: IUserService;
begin
  IUser := (HTTPRIO1 as IUserService);
  cdsUserMaint.Active := false;
  cdsUserMaint.XMLData := IUser.GetUserList (edtQryByName.Text);
  cdsUserMaint.Active := True;

```



```

    IUser: = nil;
    btnUpdate.Enabled: = true;
end;

procedure TFormMain.btnUpdateClick (Sender: TObject);
var
    IUser: IUserService;
begin
    IUser: = (HTTPRIO1 as IUserService);
    if IUser.UpdateUserData (cdsUserMaint.XMLData) = 0 then
        WebApplication.ShowMessage ('更新成功!', smAlert, '操作提示')
    else
        WebApplication.ShowMessage ('更新失败!', smAlert, '操作提示');
    IUser: = nil;
    btnQryByNameClick (nil);
end;

procedure TFormMain.IWAppFormCreate (Sender: TObject);
var
    IUser: IUserService;
    i, count: integer;
    aDeps: TStringDynArray;
begin
    count: = 0;
    IUser: = (HTTPRIO1 as IUserService);
    aDeps: = IUser.GetDepList (count);
    for i: = 0 to count-1 do
        dcbcbDep.Items.Add (aDeps [i]);
    end;

    procedure TFormMain.IWDBGrid1Columns0Click (ASender: TObject;
        const AValue: String);
    begin
        cdsUserMaint.Locate ('ID', AValue, []);
    end;

    procedure TFormMain.btnExitClick (Sender: TObject);
    begin
        WebApplication.Terminate ('感谢使用,再见!');
    end;

    procedure TFormMain.IWCheckBox1Click (Sender: TObject);
    begin
        if IWCheckBox1.Checked then
            self.Background.URL: = 'http://ly/ws/B003.jpg'
        else self.Background.URL: = '';
    end;

end.

```

我们在 Web Form 中除了可以像普通窗体那样设置背景颜色，还可以像普通网页那样设置背景图片。不过方法是通过设置对象的属性实现的。

```
self.Background.URL: = 'http://ly/ws/B003.jpg' // 这里 self 是 TFormMain
```

于是这个 Web 应用程序的运行结果如图 8-54 所示。它不仅成功调用了 Web Service，还增加了漂亮的背景图案。

读者注意图 8-54 中 IE 的地址栏，可以发现这是一个实际运行的 Web 应用程序截图。我们可以用一个固定的网址来调用 Web Form。那么，这个 Web 应用程序是如何部署到 Web 服务器上的呢？

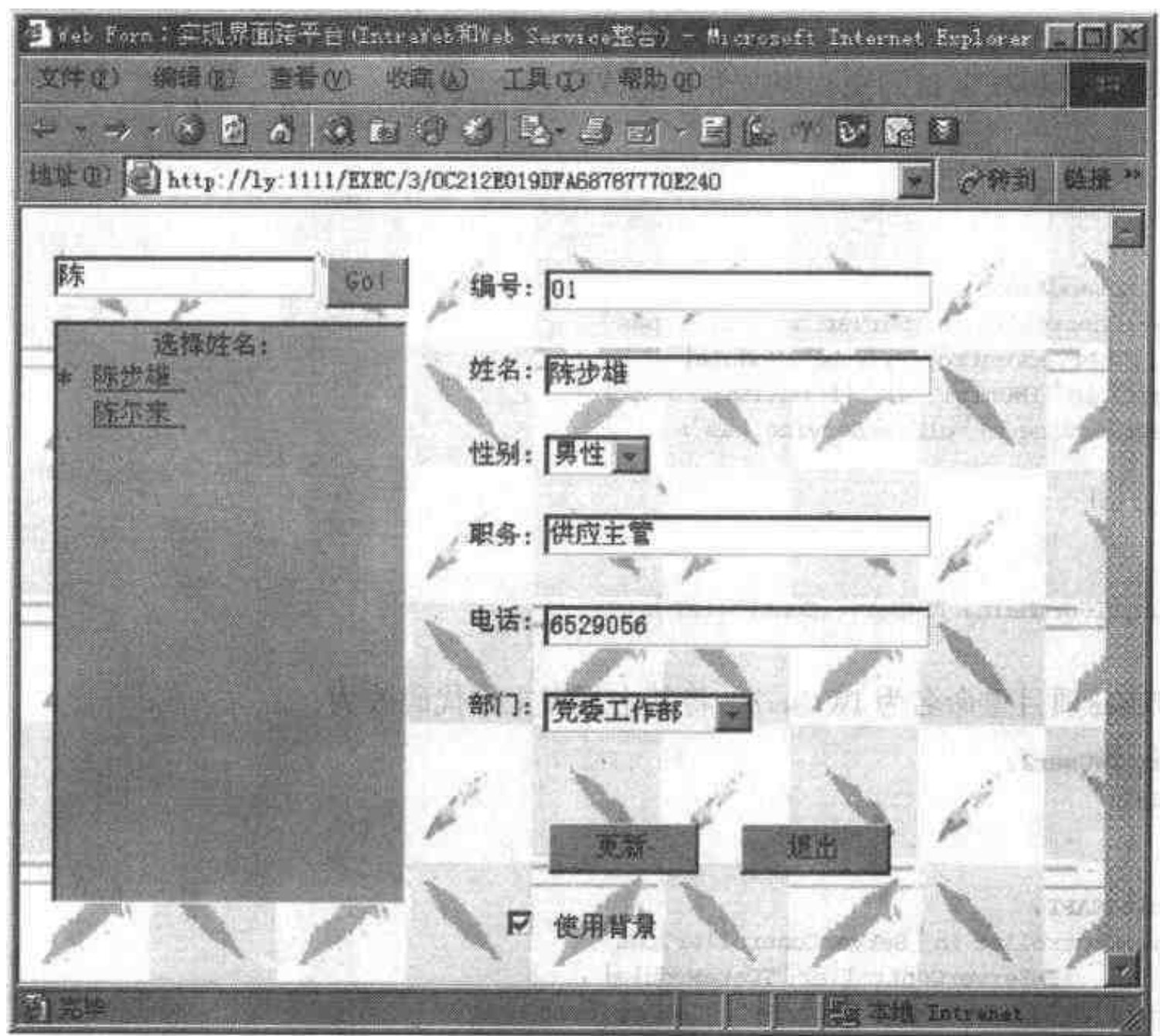


图 8-54 IE 中 Web Form 不仅成功调用了 Web Service，还增加了漂亮的背景图案

实际上部署 IntraWeb 的 Web 应用程序很简单。首先我们将调试好的可执行文件复制到 IIS 的虚拟目录下，运行该程序，待其内部的 Web Server 启动后就可以了。

客户端用户只要在浏览器的网址栏输入 `http://<服务器网址>:端口号` 就可以浏览到 Web Form 页面。这里的端口号需要在 ServerControler 中的 port 属性中设置。如果没有手工设置，则缺省的属性值为 0，如图 8-50 所示。这样，运行程序时，实际端口号将是随机的。

所以，我在示例程序中将 port 属性设为 1111，这样就可以在浏览器网址栏中输入以下地址：

`http://ly:1111/`

如果 Web 服务器中有多个 Web 应用，为避免端口号冲突，可以设置 ServerControler 中的 StartCmd 属性。如果设为 /UserApp，就需要在浏览器网址栏中输入以下地址来开启网页：

http: //ly: 1111/UserApp

对于中小型的 Web 应用来说, IntraWeb 的 Stand Alone Application 类型的 Web 应用程序简单易用, 调试方便, 是一个不错的选择。但是对于用户访问量大的大型应用, 创建 ISAPI 或 Apache DSO 类型的 Web 应用程序则比较适合。

将 Stand Alone Application 类型的 Web 应用程序改为 ISAPI 或 Apache DSO 类型的 Web 应用程序对于 IntraWeb 而言并不困难。下面就演示如何把前面的程序改为 ISAPI 类型。

打开原来 IWUser 项目的主程序文件, 其代码如下所示:

```
program IWUser;
{PUBDIST}

uses
  IWInitStandAlone,
  ServerController in 'ServerController.pas'
    {IWServerController: TDataModule},
  IWUnit1 in 'IWUnit1.pas' {formMain: TIWAppForm},
  uIUserService in 'uIUserService.pas';

{$R *.res}

begin
  IWRun (TFormMain, TIWServerController);
end.
```

将 IWUser 项目重命名为 IWUser2, 将其主程序文件代码改为:

```
library IWUser2;
{PUBDIST}

uses
  IWInitISAPI,
  ServerController in 'ServerController.pas'
    {IWServerController: TDataModule},
  IWUnit1 in 'IWUnit1.pas' {formMain: TIWAppForm},
  uIUserService in 'uIUserService.pas';

{$R *.res}

begin
  IWRun (TFormMain, TIWServerController);
end.
```

其中粗体部分是改动的地方, 共有两处。然后重新编译, 并复制到 IIS 的虚拟目录下。在浏览器网址栏中输入以下地址来开启网页:

http: //ly/ws/iwuser2.dll/

一切 OK。说明 ISAPI 类型的 Web 应用程序转换成功。连我都感到简单得难以置信! 看来 IntraWeb 的确是进行 Web 开发的低成本、高效率的选择。

第9章 深入浅出 VCL (上)

9.1 Delphi 对象的基础: VCL

在采用面向对象的语言编程时,首先要解决的问题有程序中需要设计的类、类所需完成的功能、每个类的继承关系及类与类间的关系,然后才进行类的框架设计,设计类的接口及每个类的属性、方法和事件,最后才是编码。在这几个工作中,前两个工作是程序设计的难点和重点,进行这项工作前,不仅要求设计人员对所要完成的工作需求和要达到的目标十分清楚,而且要对 Delphi 的 VCL 类和对象有较深的理解,并对 VCL 各底层类的继承关系十分熟悉。这样才能合理地安排程序类的组织关系和确定类的继承关系,才能充分利用类的属性、方法和事件的不同来合理设计程序的接口,简化程序最后的组装工作,并使类间的耦合关系合理而有利于类的更新和修改。总之,熟练掌握 Delphi 的类的相关概念和 VCL 类的组织结构及类间的关系是从事 Delphi 应用程序开发的基本条件,要深刻理解掌握这些知识,并能灵活应用,需要程序员在具体的编程实践中不断实践加深才可最终水到渠成。所以说 VCL 是 Delphi 对象的基础。但掌握 VCL 不是会使用几个 VCL 控件,通过拖放实现傻瓜编程;而是要了解 VCL 的基本框架和领会 VCL 的核心组件,灵活使用 VCL 的现有资源,实现编程的最佳效率。

9.1.1 VCL 的层次结构

Delphi 应用程序的开发者主要是通过 VCL (一个包含组件集合的体系框架) 来建立应用程序;而对于组件的编写者来说, VCL 则代表了扩展的类,通过继承,将 VCL 中可扩展类中的一些功能函数集成到自己的控件中。这样既扩展了 VCL 库,同时又可将该控件作为一个可扩展的祖先类供其他人在此基础上继续开发。

图 9-1 显示了 VCL 体系结构中的一些基本的类。最顶端是 TObject,它是 Delphi 中所有类的抽象基类。TPersistent 类是直接来自 TObject 类继承下来的。凡是从 TPersistent 继承下来的对象都具有进行流操作的能力。流是一个以二进制数据形式封装在存储介质(例如内存或磁盘文件)中的对象。因为 Delphi 窗体文件是通过流实现的,直接来自 TPersistent 继承下来的 TComponent 使得所有控件具有生成 Form 文件的能力。

图 9-1 中 4 个带底纹的类代表了创建新类的 4 个基类。了解这些基类的继承关系对设计我们自己的类并正确选择父类是很有帮助的。

TComponent 类也是 4 个基类中首个可被用来创建新组件的基类,非可视化组件也是从 TComponent 继承下来的。TControl 类作为一个虚拟类实现了可视组件一些公用的特性,从 TControl 类继承下来的组件我们通常称之为控件。注意到图 9-1 中有两种基本的可视化控件: graphic controls 和 windowed controls,它们的代表分别是 TGraphicControl 类和 TWinControl 类。它们之间主要的区别在于 TGraphicControl 继承下来的控件没有窗口句柄,因而也没有输入焦点。窗口控件进一步分可分为两类。一类是直接来自 TWinControl 继承下来的通过 Windows 内部控件实现

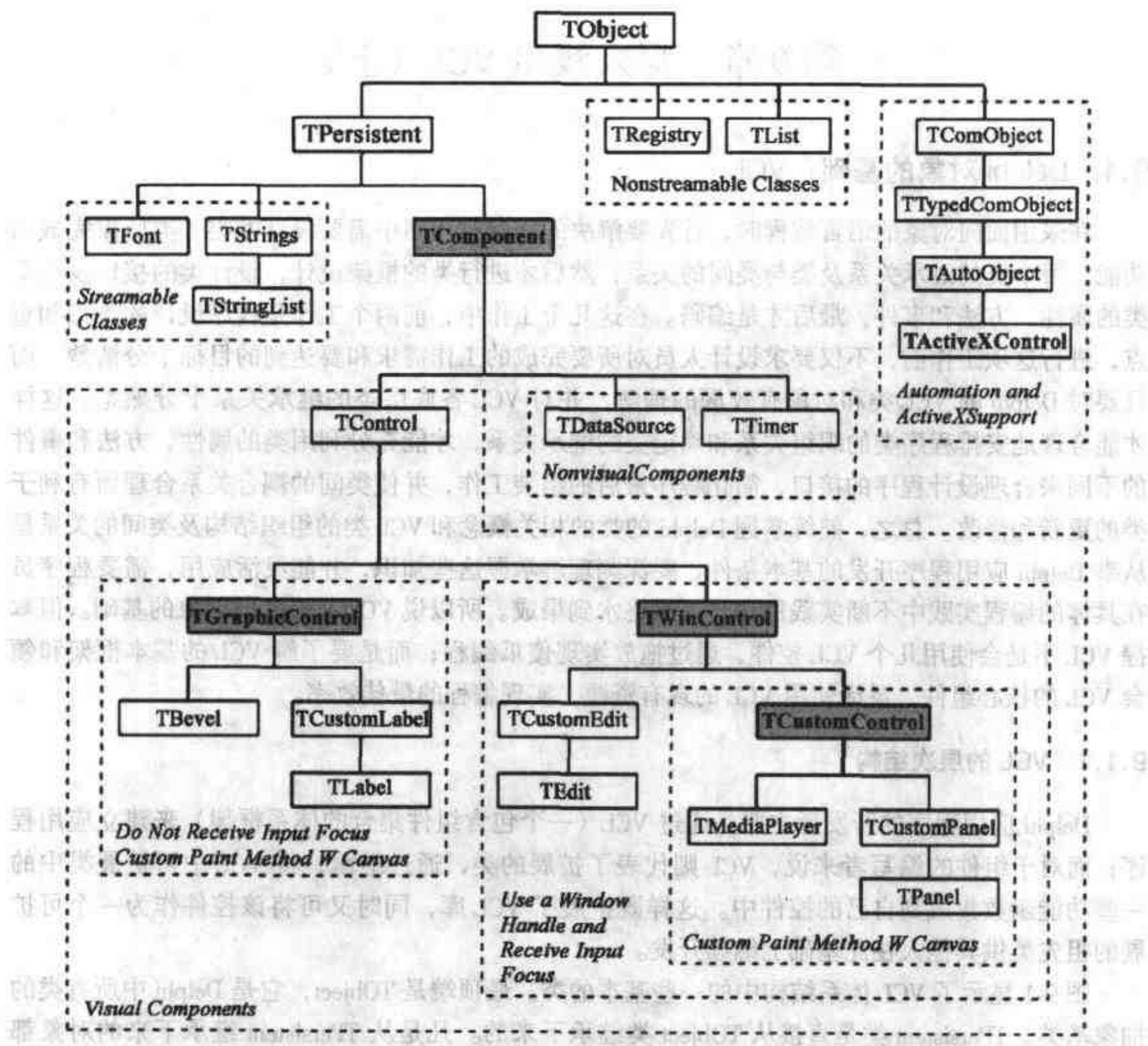


图 9-1 VCL 的层次结构

的控件，它能够自己自动实现重绘功能。例如编辑框便是一个直接从 TWinControl 继承下来的控件。而另一类 TCustomControl 类则指那些需要窗口句柄控件但未封装提供重绘功能的 Windows 控件。

从层次结构上，我们还可以将 VCL 分为三个主要区：组件区、通用对象区和异常区。尽管 VCL 基本上是组件的集合，但在 Delphi 对象中，除 VCL 的组件以 TComponent 为基类外，还有许多类是以 TObject、TPersistent 等为基类的，如 TFont、TPicture、TList 等。由于它们没有继承自 TComponent，这些非组件类分布在 VCL 的通用对象区和异常区，通常称之为对象（object），尽管这不是一个准确的定义。

在 VCL 中常用的非组件类包括：

- 与图形有关的对象——TBitmap、TBrush、TCanvas、TFont、TGraphic、TGraphicsobject、TIcon、TMetafile、TPen 和 TPicture。
- 与流/文件有关的对象——TBlobStream、TFileStream、THandleStream、TIniFile、TMemoryStream、TFile、TReader 和 TWriter。
- 列表和集合——TList、TStrings、TStringList、TCollection 和 TCollectionItem。
- 与 COM 有关的类——这是 Delphi 编程的一个重要领域。
- 异常类——这是继承自 Exception 类的类。

上述的这些非组件类主要有两个用处：第一是非组件类可以定义组件属性的数据类型，如图像组件（TGraphic 对象）的 Picture 属性或列表框（TString 对象）的 Items 属性。这些类一般继承自 TPersistent，所以是流式（streamable）的，可以有子属性甚至事件。

第二是可以直接使用。在用户编写的 Delphi 代码中，可以分配和处理这些类的对象。这样做有很多用处，包括在内存中存储属性值的拷贝并在不改变原组件的情况下改动它，存储值的列表、编写复杂算法等等。

9.1.2 组件的继承关系

TComponent 类是所有 Delphi 组件的基类。而我们前面说过，TObject 类是 Delphi 类的基类，即所有的 Delphi 类都是从 TObject 类继承的，例如：

```
TMyObject = Class (TObject)
TMyObject = Class
```

这两条语句所声明的对象 TMyObject 是一样的，都是由 TObject 对象继承而来的。

Delphi 定义的类很多，但这些类按它的基类可分为几种，即 TObject 分支类、TException 分支类、TPersistent 分支类、TComponent 分支类、TControl 分支类、TWinControl 分支类和 TGraphicControl 分支类。从结构上看，这几个基类构成了 Delphi 类的主干，由这些主干再派生 Delphi 丰富多彩的各专用类，所有的这些类构成了 Delphi 复杂而功能强大的类库。这些主干类的继承关系可参见后面的图 9-2。

TComponent 类处在组件层次结构的顶层。TComponent 类是所有 VCL 组件的基类，从该类派生的 VCL 类又分为非可视组件类（Nonvisual Component）和可视组件两大类（Visual Component）。非可视组件就是一些用于完成某种与程序界面无关的功能的组件，比如像 TQuery、TTable 等组件。在程序设计期，组件可放置到窗体上并通过 Object Inspector 窗口进行组件的属性、事件等的设置，以使组件的应用更方便；而在程序运行期，组件在窗体上并不可见，在运行期，它往往要通过其他的一些可视组件来完成程序与用户信息的沟通和传输。与非可视组件相反，在运行期，在窗体上可见的组件称之为可视组件，如 TEdit、TListBox 和 TImage 等组件。可视组件主要用于完成与程序的输入、输出接口相关的功能，它一方面使用户的操作意图及程序运行所需的信息能够准确方便地传给程序，另一方面组件能将程序的运行结果及时地反馈给用户，使用户能根据运行的结果控制程序继续正确运行。对于可视组件，它们又被分成两大类，一类称为 Windows 类可视组件（有窗口句柄的可视组件），而另一类称为非 Windows 类可视组件（没有窗口句柄的可视组件）。这两类的主要差别在于是否有窗口句柄。对于 Windows 类组件，每个组

件本身就是一个实实在在的 Windows 窗口，它是 `TWinControl` 或其子类的派生类，并拥有输入焦点和自己的消息队列管理函数。在 VCL 可视组件中，大多数的组件都属于这一类，如 `TEdit`、`TButton`、`TComboBox` 和 `TListBox` 等。非 Windows 组件虽然从组件的外观上无法将它与 Windows 可视组件区分开来，但在编程使用上却有较大的区别。非 Windows 可视组件一般是 `TGraphicControl` 类或其派生类的子类，它没有窗口句柄、输入焦点和自己独立的消息队列管理函数，因为它不是一个 Windows 窗口，只是利用它所寄生的窗口来显示有关的信息和组件外观。由于没有窗口句柄，非 Windows 组件也就不可能有输入焦点和自己的消息队列管理函数，它是利用组件所寄生的窗体（如：`TForm` 及其派生类）或容器组件（如：`TPanel` 等容器组件）的消息队列管理函数来处理消息而调用组件的消息处理函数，属于这一类的组件有 `TImage`、`TShape` 等。

为了更清楚地说明 Delphi 类的组织关系，下面是简化的 VCL 主要基础类关系结构图，如图 9-2 所示。

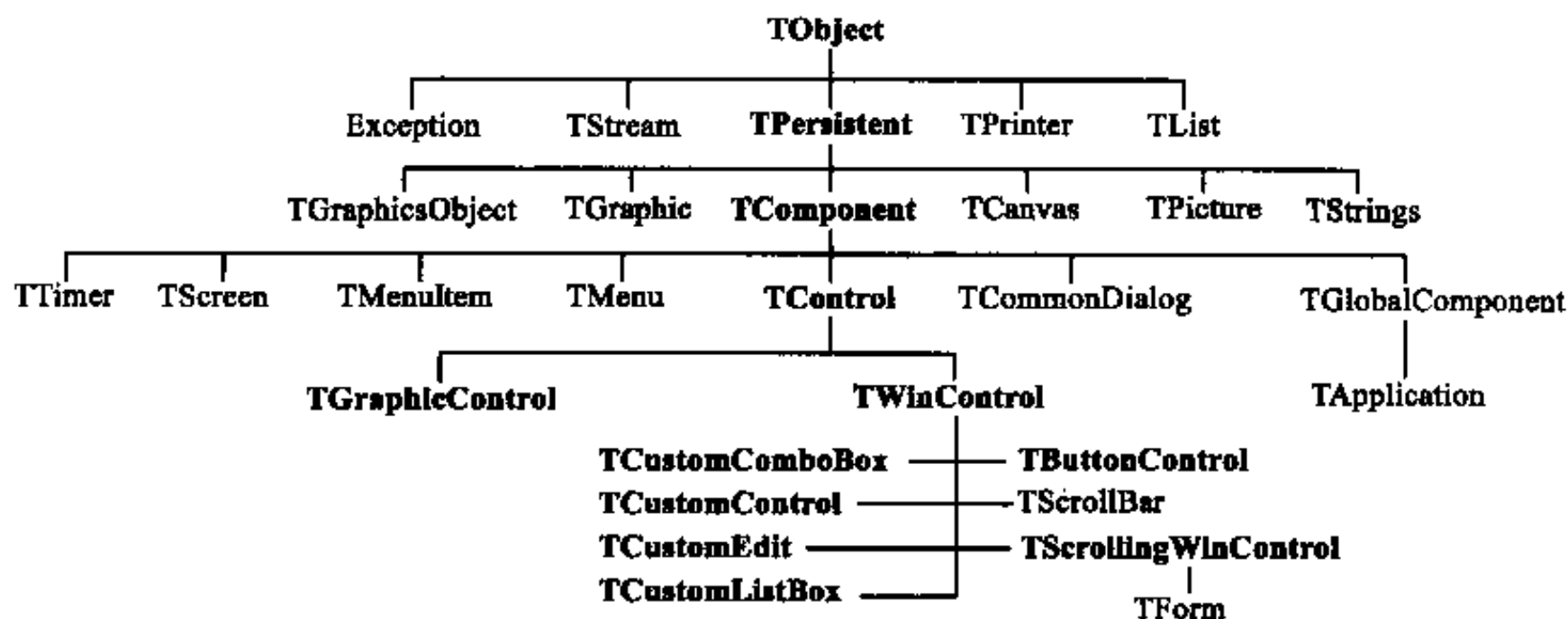


图 9-2 Delphi 主要基类及其主要派生类的继承关系

9.2 TObject: 所有对象的根

`TObject` 类定义在 `System.pas` 单元中，它是所有 Delphi 类的基类，即 Delphi 的任何类都是 `TObject` 类的派生类。`System.pas` 单元是由编译器自动加入 Delphi 用户的项目中的，用户自己无法修改、删除 `System.pas` 中的任何东西，也无法手工加入 `System.pas` 单元（否则系统会报错）。这就是说，`TObject` 是属于 Delphi 编译器的特性。

Delphi 的核心是类的层次结构。系统中的每一个类都是 `TObject` 的子类，所以整个类的层次结构只有一个根类。这允许用户在系统中用 `TObject` 数据类型替代任何类的数据类型。例如，事件处理程序通常有一个 `TObject` 类型的 `Sender` 参数，这意味着 `Sender` 对象可属于任何类，因为每个类都是从 `TObject` 类派生出来的。这种方式的一个典型缺陷是，在对象上进行操作时用户必须知道它的数据类型。事实上，`Object` 类型的变量或参数只能应用于由 `TObject` 定义的对象方法和属性。例如，如果该变量或参数引用一个 `TButton` 类型的对象，用户将不能直接访问它

的 Caption 属性。解决问题的方法是，使用 is 和 as 这样的类操作符来检查数据类型并进行实际的类型转换。例如，如果 TObject 类型的 Sender 参数引用了一个按钮对象，用户可以这样写：

```
(Sender As Tbutton).Caption
```

这种方法的好处是，当类型失败时，Delphi 会引起异常，这样可以保证代码的安全。

根据子类可赋值给基类的规则，所有的对象都可赋值给 TObject 类型的变量或形参，因此对象在函数中的传递是十分方便和灵活的。在事件的 Sender 中传入的是引发事件的对象，因此在事件处理代码中可根据引发事件对象的不同而执行不同的代码，这就为多个组件共用一个事件处理函数提供了基础。但是，TObject 类只声明了所有类都共有的一些最基本的方法，而派生类往往有许多专用的方法，要在事件处理函数中调用或访问派生类的专有方法或属性，则必须将传入的对象按对象所属的类进行强制类型转换，强制类型转换要求类型必须匹配，否则会引发类型转换异常。在类型转换中为保证类型匹配正确，需要调用对象的有关方法来进行对象所属类的判断，这些与判断对象所属类有关的最底层的方法就声明在 TObject 类中。

对于 Delphi 类或组件设计程序员，了解和掌握 Delphi 的 TObject 类是至关重要的。TObject 包括了诸如实例初始化、实例析构、RTTI、消息分发等相关的实现方法，它在 System 单元中被声明如下：

```
Type
TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance (Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs (const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom (AClass: TClass): Boolean;
    class function MethodAddress (const Name: ShortString): Pointer;
    class function MethodName (Address: Pointer): ShortString;
    function FieldAddress (const Name: ShortString): Pointer;
    function GetInterface (const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry (const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException (ExceptObject: TObject;
        ExceptAddr: Pointer): HRESULT; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch (var Message): virtual;
    procedure DefaultHandler (var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
end;
```

从 TObject 的声明看, 该类的大多数方法前都加上了 class 关键字, 即方法被声明为类方法。类方法与普通的方法不同, 普通的方法只有在对象被实例化后才可调用, 否则会因对象未被创建而引发异常, 这种异常所表现出的特征往往是莫名其妙的。如果跟踪程序就会发现, 这种因对象未实例化而造成的异常往往是在调用对象的方法时引发访问无效内存的提示, 并有可能造成死机等; 并且在出错时很难被发现, 因为程序的逻辑是正确的。而类方法的调用却不需要对象实例化, 即对象被声明后就可调用类方法, 因此类方法的编写有一定的约束, 即不能访问类所声明的变量。因为类未实例化, 变量的存储空间还未分配。在 TObject 类中声明的类方法主要是一些与类的类型、方法地址、类继承等这些最基础的操作相关的方法。除类方法外, 在 TObject 中声明最多的是一些与对象的实例化和对象销毁有关的虚方法, 即在方法的后面带有 virtual 关键字。在 TObject 中声明的方法中, 有相当一部分是由 Delphi 内部调用的, 而在应用程序中一般不需要直接调用这些方法, Delphi 内部使用的方法有: ClassInfo、CleanUpInstance、FieldAddress、FreeInstance、InitInstance、MethodAddress、MethodName 和 NewInstance。这些 Delphi 内部调用的方法在一般编程中基本上不会调用, 除非有特殊需要的场合或为编写更灵活的应用程序才有可能使用这些函数, 但是理解 FieldAddress、MethodAddress 和 MethodName 这三个函数对理解 Delphi 可视化编程的实现机理是很有帮助的。在 TObject 类中, ClassNameIs 和 ClassType 二个方法主要用来判断类的类名和类的继承关系, 在实际编程中一般用 As 和 Is 两个操作符而不在程序中直接调用这二个方法, 但是, 如果要判断一个类是否是另一个类的祖先类时, 则可用 InheritsFrom 来判断, 这是用 Is 和 As 操作符所无法替代的。DefaultHandler 是类缺省的消息处理过程, 它是由类的消息分发过程 Dispatch 来调用的, 而 Dispatch 也是由 Delphi 内部调用的。GetInterface 和 GetInterfaceEntry 与 COM 对象所支持的接口有关, 在 COM 对象应用程序中, 一般也不直接调用这两个函数, 取而代之的是用 As 操作符。GetInterfaceTable 用于取对象所支持的接口表, 这也是由 Delphi 内部来使用的。而 SafeCallException 用来处理 COM 对象的异常; 由 COM 对象的基类来重载该虚方法, 程序并不直接调用该方法。在其中的方法中, 除一些与对象的创建和销毁有关的函数外, 主要用来获取与对象相关的一些信息, 这些函数正是我们在编程中需要调用的。

为了说明这些非 Delphi 内部使用的方法, 下面我们用一个演示 TObject 方法的程序例子来说明。在示例程序 9-1 中, 我们将窗口上所有 VCL 组件的 OnClick 事件都指到同一个事件处理函数, 在 OnClick 事件处理函数中, 将引发事件所属的类、实际的字节数、父类名等有关信息在 ListBox1 中列出, 如图 9-3 所示。

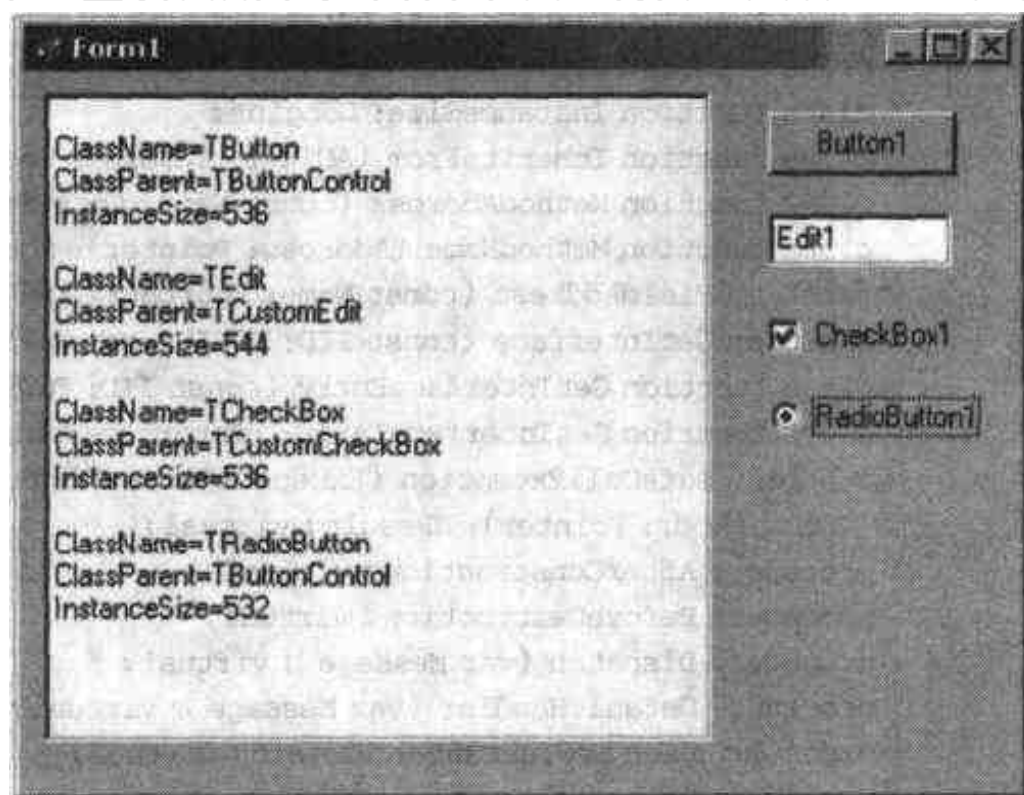


图 9-3 演示 TObject 方法的程序运行界面

示例程序 9-1 演示 TObject 方法的程序

```
unit TObjectMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ExtCtrls, StdCtrls;

type
  TForm1 = class (TForm)
    ListBox1: TListBox;
    Button1: TButton;
    Edit1: TEdit;
    CheckBox1: TCheckBox;
    RadioButton1: TRadioButton;
    procedure OnClick (Sender: TObject);
    procedure FormCreate (Sender: TObject);
  private
    {Private declarations}
  public
    {Public declarations}
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.OnClick (Sender: TObject);
Var
  Str: String;
begin
  Str: = '';
  ListBox1.Items.Add (Str);
  Str: = Format ('ClassName = %s', [Sender.ClassName]);
  ListBox1.Items.Add (Str);
  Str: = Format ('ClassParent = %s', [Sender.ClassParent.ClassName]);
  ListBox1.Items.Add (Str);
  Str: = Format ('InstanceSize = %d', [Sender.InstanceSize]);
  ListBox1.Items.Add (Str);
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
  ListBox1.Items.Clear;
end;

end.
```

在例子中，如果要在 ListBox1 中列出引发事件类的所有继承关系，只需要改写 OnClick 事

件处理函数就可实现，修改后的代码如下：

```
procedure TForm1.OnClick (Sender: TObject);
Var
  Str: String;
  TmpClass: TObject;
begin
  TmpClass := Sender;
  While TmpClass.ClassParent <> nil do
  Begin
    Str := TmpClass.ClassName;
    ListBox1.Items.Add (Str);
    TmpClass := TmpClass.ClassParent;
  End;
End;
```

下面的例子是一个如何获取某组件的祖先类的类型和祖先类的有关属性，该例子分别放置了 TButton 和 TListBox 组件在 Form 上，当用户单击按钮时，这个按钮的类名和它的所有祖先类的类名都将加入到 Listbox1 中。程序如下：

```
procedure TForm1.Button1Click (Sender: TObject);
var
  ClassRef: TClass;
begin
  ListBox1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    ListBox1.Items.Add (ClassRef.ClassName);
    ClassRef := ClassRef.ClassParent;
  end;
end;
```

当用户点击按钮时，列表框中包含了以下的信息：

- TButton
- TButtonControl
- TWinControl
- TControl
- TComponent
- TPersistent
- TObject

9.3 TPersistent：持久对象

TPersistent 类在 Delphi 类中的地位不亚于 TObject 类，该类定义于 Classes 单元中，Delphi 中定义的类几乎都是由该类派生面来的。TPersitent 之所以在 Delphi 类中居于如此重要的地位是因为它是 Delphi 可视化编程的基础。该类实现了对象公布 (published) 属性的存取，即在该类及其派生类中声明为 published 的属性、方法和事件等可在设计期时显示在 Object Inspector 窗中，能在 Object Inspector 中对对象的 published 属性进行设计期的设计，并可将设置的值存到窗体或

数据模块的 DFM 文件中。当 Delphi 重新打开原来保存的窗口时，对象的设置值将被 Delphi 从 DFM 文件中读出。在程序运行期，对象将被初始化为设计期所设置的状态。TPersistent 类声明十分简单，只有 7 个方法，具体定义如下：

```
{ $M+ }

TPersistent = class (TObject)
private
    procedure AssignError (Source: TPersistent);
protected
    procedure AssignTo (Dest: TPersistent); virtual;
    procedure DefineProperties (Filer: TFiler); virtual;
    function GetOwner: TPersistent; dynamic;
public
    destructor Destroy; override;
    procedure Assign (Source: TPersistent); virtual;
    function GetNamePath: string; dynamic;
end;

{ $M- }
```

看完了 TPersistent 类的声明，大家可能会觉得奇怪，这简单的 7 个方法如何完成类的属性、方法和事件的存取呢？实际上完成存取工作的代码并没定义在 TPersistent 类中，而是由 Delphi 另行定义，如 TReader 和 TWriter 等对象，这些对象都是以 TPersistent 为服务对象的。

从 TPersistent 类的声明看，它与 TObject 的声明方式和结构基本相同，但为什么 TObject 却没有 published 属性、方法和事件的设计期的存取功能呢？如果大家仔细看一下 TPersistent 的声明就会发现，在 TPersistent 的声明中多了 M 编译开关，{ \$M+ } 与 { \$TYPEINFO ON } 作用是一样的，而 { \$M- } 与 { \$TYPEINFO OFF } 相同。当一个类在 { \$M+ } 和 { \$M- } 间声明时，程序编译器将为类生成与 RTTI (Runtime Type Information) 相关的代码来完成类的 published 的属性、方法和事件的存储工作。并且该类子类的 published 属性、方法和事件也具有存取特性。如果一个类或其祖先类都没有在 { \$M+ } 和 { \$M- } 中声明，则该类不能有 published 的属性、事件和方法。

对一般编程而言，TPersistent 的 Assign 方法较为常用，它主要完成两个对象属性的复制。对于两个对象，如：

```
Var
    Obj1, Obj2: TFont;
begin
    Obj1 := Obj2;
end;
```

并不是像记录数据类型一样完成了两个对象属性的复制，而是 Obj 被设成为 Obj2 的对象引用。因此必须在 Assign 方法中完成对象的 published 属性、方法和事件的逐个复制。

Assign 方法在 TPersistent 类中声明为虚方法，以便允许每个派生类定义自己的复制对象方法。如果派生类没有重写 Assign 方法，则 TPersistent 的 Assign 方法会将复制动作交给源对象来进行：

```

procedure TPersistent.Assign (Source: TPersistent);
begin
  if Source < > nil then Source.AssignTo (Self)
else AssignError (nil);
end;

```

由此可见, Assign 方法实际上是调用 AssignedTo 方法来实现的, 因此 TPersistent 的 Assign 方法很少被派生类所覆盖, 但 AssignTo 却常被派生类根据需要覆盖。

如果由 AssignedTo 方法来实现复制, 那么必须保证源对象的类已经重写了 AssignedTo 方法, 否则将抛出一个 AssignError 异常:

```

procedure TPersistent.AssignError (Source: TPersistent);
var
  SourceName: string;
begin
  if Source < > nil then
    SourceName := Source.ClassName else
    SourceName := 'nil';
  raise EConvertError.CreateResFmt (@SAssignError, [SourceName, ClassName]);
end;

```

Assign 方法和 AssignedTo 方法常用于对象的克隆。

GetPathName 方法是 Delphi 内部调用的, 用于取得对象名以显示在 Object Inspector 中。GetOwner 用于返回对象的所有者, 它往往与 GetPathName 一起使用, GetOwner 在 TPersistent 中只返回 nil, 该方法在 TPersistent 类中声明而不是在 TComponent 类中声明, 是为了使一些由 TPersistent 派生的类可在 Object Inspector 中显示出来, 如 TCollection 等类。对于一般应用程序设计来说, 不需要直接调用这两个方法。

在 TPersistent 中声明的 DefineProperties 方法为非 published 属性存取提供了接口。对于从 TPersistent 派生的子类, Delphi 编译器在编译时自动为类的 Published 的属性加入与存取相关的代码, 而对于其他的非 published 的属性要存入 DFM 文件或流中; 需要在派生类中覆盖 DefineProperties 方法来完成这些属性值的存取操作。具体调用方法如下:

```

Type
  TSampleObject = Class (TPersistent)
  Private
    FMyProperty: Integer;
    procedure LoadCompProperty (Reader: TReader);
    procedure StoreCompProperty (Writer: TWriter);
  Protected
    procedure DefineProperties (Filer: TFiler); override;
  Public
    MyProperty: integer read FMyProperty Write FMyProperty;
  End;

```

Implementation

```

procedure TSampleObject.LoadCompProperty (Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyProperty := Reader.ReadComponent (nil);

```

```

end;

procedure TSampleObject.StoreCompProperty (Writer: TWriter);
begin
  Writer.WriteBoolean (MyProperty < > nil);
  if MyProperty < > nil then
    Writer.WriteComponent (MyProperty);
end;

procedure TSampleObject.DefineProperties (Filer: TFiler);
function DoWrite: Boolean;
begin
  if Filer.Ancestor < > nil then {祖选类是否有同名属性}
  begin
    if TSampleObject (Filer.Ancestor).MyProperty = nil then
      Result := MyProperty < > nil
    else if MyProperty = nil or
      TSampleObject (Filer.Ancestor).MyProperty.Name < > MyProperty.Name then
      Result := True
    else Result := False;
  end
  else {祖选类中无同名属性}
    Result := MyProperty < > nil;
  end;
begin
  inherited; {调用父类方法以保存相关的属性}
  Filer.DefineProperty ('MyProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;

```

在上面的代码中，LoadCompProperty 和 StoreCompProperty 方法是实现属性存取的具体代码，最后传给 Filer.DefineProperty 方法而由 Filer 对象调用来完成属性的存取操作。

在对象创建时，Delphi 如何能将存储在 DFM 文件中的组件的事件和属性值恢复呢？实际上 Delphi 利用了 TObject 中三个基础的方法来实现对象属性的存取恢复设置工作。我们先来看一个 MethodAddress 方法，该方法能返回指定方法名的方法在内存中的指针，利用该指针即可调用该方法，例如：

```

type
  TForm1 = class (TForm)
    Button1: TButton;
    procedure Button1Click (Sender: TObject);
  private
  public
  end;

implementation
{$R *.dfm}
procedure TForm1.Button1Click (Sender: TObject);
begin
  Caption := 'Button1 Click';
end;

```

当 Button1 收到按钮按下消息时，它要完成对声明在 TForm1 中的 Button1Click 事件处理函数

的调用，调用过程的实现类似于下面的代码。

```
Var
    Method: TMethod;
    Event: TNotifyEvent;
Begin
    Method.Code = Owner.MethodAddress (' Button1Click ');
    Method.Data = Self;
    Event := TNotifyEvent (Method);
    Event (Self)
End;
```

Button1 先根据要调用的事件处理函数名而取得函数的地址，然后调用该地址所指的函数来实现。TMethod 数据结构是 Delphi 声明的专门用于存放 MethodAddress 返回结果的。函数 MethodName 的作用刚好与 MethodAddress 的作用相反，在将类的属性存到 DFM 文件时，Delphi 调用该方法以取得函数名，最后将函数名存到 DFM 文件中。FieldAddress 的作用与 MethodAddress 相类似，当对象由 DFM 创建时，Delphi 调用该方法与创建的对象进行联系以恢复对象的设置。在实际编程中，一般无需直接调用这三个底层的方法，但是理解这三个方法有利于理解 Delphi 可视化设计实现的底层机理。

在一个 TForm 上放置一些组件，为这些组件的事件设置一些事件处理函数后再看 TForm 单元的代码，我们会发现在窗体上的组件和事件处理函数被声明在 TForm 的最前面，且未加 private、protected、public 或 published 关键字，那么这些声明是属于什么访问限制呢？对 Delphi 面向对象编程较熟的人可能会按对象声明的缺省限制来理解，即 public 访问限制，而实际上大多情况是属于 published 限制，这主要是大多数 Delphi 类都是从 TPersistent 派生来的，而 TPersistent 是在 \$M+ 编译开关下编译的。在 \$M+ 状态下声明的类及其派生类，Delphi 对在类的 private、protected 和 public 等之前声明的属性和方法当做 published 限制。换句话说，也就是在对象的私有属性或方法声明前声明的都属于 published，且要存到 DFM 文件中，并且同 TForm 一同被创建。需要注意的是，并不是所有从 TPersistent 继承的类都是在 \$M+ 下编译的，如 TList、TStream 等却是在 \$M- 下编译的。

9.4 TComponent: 组件对象

9.4.1 概述

组件是 Delphi 应用程序中的核心元素。当用户编写程序时，首先要选择一些组件并定义它们的相互作用。对于大多数 Delphi 编程来说，这就是要做的全部工作。

在 Delphi 中有不同种类的组件，大部分组件都包括在 Components 面板中，但有一些（包括 TForm 和 TApplication）例外。从技术上讲，组件是 TComponent 类的子类。它们可以以流的形式存储在 DFM 文件中（因为它们都继承自 TPersistent 类，该类可以实现流），并且它们可以拥有 published 属性和可视化处理的事件。

组件的层次结构通常被划分为三个区，如图 9-4 所示。这些组都用相似的内部结构来说明组件。

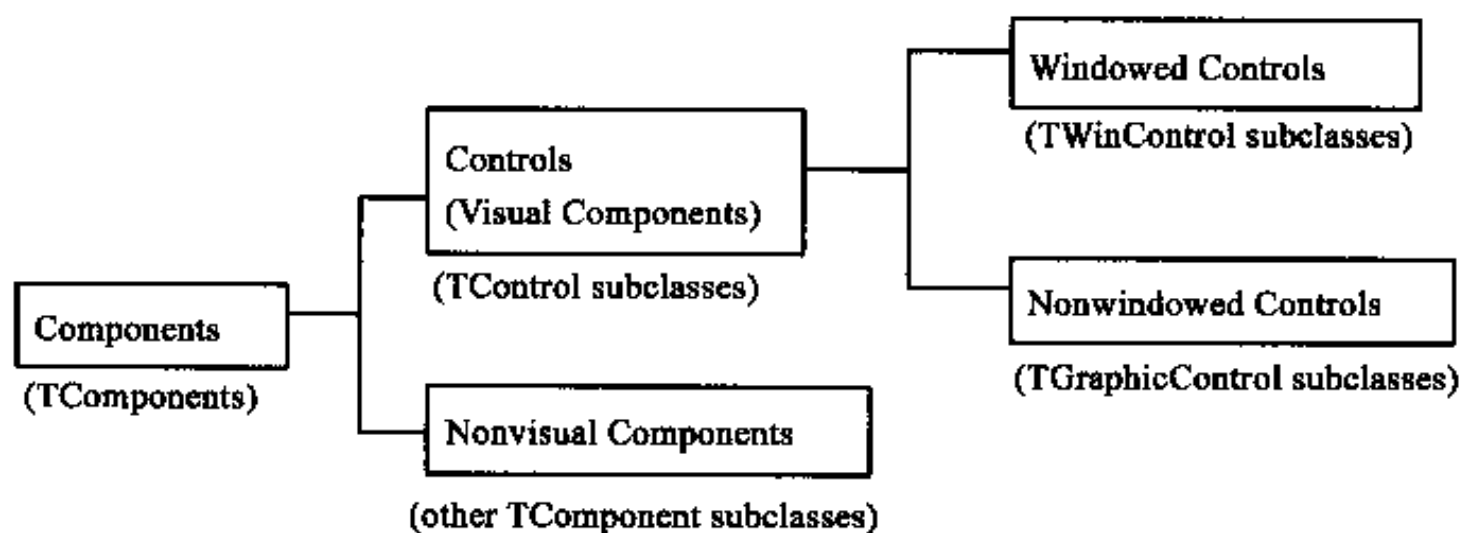


图 9-4 组件组的结构图

- 基于窗口的控件（window-based control）或窗口控件（windowed control）是基于系统窗口的可视化组件。从技术角度来讲，表明这些控件都有窗口句柄。从用户的角度来看，窗口控件可以接受输入焦点并可以包含其他控件。这是 Delphi VCL 中最大的一组组件。
- 图形控件（graphic control）或叫非窗口控件（nonwindowed control）是不基于窗口的可视化组件。所以，它们没有窗口句柄，不能接受焦点，不能包含其他控件。这些控件继承自 TGraphicControl 并由它们的父窗体显示，发送给与鼠标有关的其他事件。非窗口控件的例子有 Label 和 SpeedButton 组件。该组中控件不多，但在最小化使用系统资源方面起着关键作用，特别是对于那些频繁使用且数量大的组件（如标签或工具条按钮）更是如此。
- 非可视化组件（nonvisual component）是控件以外的所有组件——所有从 TComponent 但不从 TControl 继承的类。在设计时，非可视化组件以图标形式出现在窗体上。运行时，其中一些组件是可见的（如标准对话框），而其他的则是不可见的（如数据库表格组件）。换句话说，非可视化组件在运行时是看不到的，尽管它们可能管理某些可以看到的東西，如对话框。

TComponent 类是所有组件的祖先类，理解 TComponent 在组件设计中具有十分重要的意义。对于非可视类组件来讲，可直接由 TComponent 类继承，若有与要开发的类相关的类，则可由这些类继承。为了透彻理解 TComponent 类的属性和方法以进行组件的开发，下面我们将详细介绍 TComponent 类。

```

TComponent = class (TPersistent, IInterface, IInterfaceComponentReference)
private
  FOwner: TComponent;
  FName: TComponentName;
  FTag: Longint;
  FComponents: TList;
  FFreeNotifies: TList;
  FDesignInfo: Longint;
  FComponentState: TComponentState;
  FVCLComObject: Pointer;
  function GetComObject: IUnknown;

```

```

function GetComponent (AIndex: Integer): TComponent;
function GetComponentCount: Integer;
function GetComponentIndex: Integer;
procedure Insert (AComponent: TComponent);
procedure ReadLeft (Reader: TReader);
procedure ReadTop (Reader: TReader);
procedure Remove (AComponent: TComponent);
procedure RemoveNotification (AComponent: TComponent);
procedure SetComponentIndex (Value: Integer);
procedure SetReference (Enable: Boolean);
procedure WriteLeft (Writer: TWriter);
procedure WriteTop (Writer: TWriter);
|IInterfaceComponentReference|
function IInterfaceComponentReference.GetComponent = IntfGetComponent;
function IntfGetComponent: TComponent;
protected
  FComponentStyle: TComponentStyle;
  procedure ChangeName (const NewName: TComponentName);
  procedure DefineProperties (Filer: TFile); override;
  procedure GetChildren (Proc: TGetChildProc; Root: TComponent); dynamic;
  function GetChildOwner: TComponent; dynamic;
  function GetChildParent: TComponent; dynamic;
  function GetOwner: TPersistent; override;
  procedure Loaded; virtual;
  procedure Notification (AComponent: TComponent;
    Operation: TOperation); virtual;
  procedure PaletteCreated; dynamic;
  procedure ReadState (Reader: TReader); virtual;
  procedure SetAncestor (Value: Boolean);
  procedure SetDesigning (Value: Boolean; SetChildren: Boolean = True);
  procedure SetInline (Value: Boolean);
  procedure SetDesignInstance (Value: Boolean);
  procedure SetName (const NewName: TComponentName); virtual;
  procedure SetChildOrder (Child: TComponent; Order: Integer); dynamic;
  procedure SetParentComponent (Value: TComponent); dynamic;
  procedure Updating; dynamic;
  procedure Updated; dynamic;
  class procedure UpdateRegistry (Register: Boolean; const ClassID,
    ProgID: string); virtual;
  procedure ValidateRename (AComponent: TComponent;
    const CurName, NewName: string); virtual;
  procedure ValidateContainer (AComponent: TComponent); dynamic;
  procedure ValidateInsert (AComponent: TComponent); dynamic;
  procedure WriteState (Writer: TWriter); virtual;
|IInterface|
function QueryInterface (const IID: TGUID; out Obj): HRESULT;
  virtual; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
|IDispatch|
  function GetTypeInfoCount (out Count: Integer): HRESULT; stdcall;
function GetTypeInfo (Index, LocaleID: Integer; out TypeInfo): HRESULT;
  stdcall;
function GetIDsOfNames (const IID: TGUID; Names: Pointer; NameCount,

```

```

        LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
function Invoke (DispID: Integer; const IID: TGUID; LocaleID: Integer;
        Flags: Word; var Params; VarResult, ExcepInfo,
        ArgErr: Pointer): HRESULT; stdcall;

public
    constructor Create (AOwner: TComponent); virtual;
    destructor Destroy; override;
    procedure BeforeDestruction; override;
    procedure DestroyComponents;
    procedure Destroying;
    function ExecuteAction (Action: TBasicAction): Boolean; dynamic;
    function FindComponent (const AName: string): TComponent;
    procedure FreeNotification (AComponent: TComponent);
    procedure RemoveFreeNotification (AComponent: TComponent);
    procedure FreeOnRelease;
    function GetParentComponent: TComponent; dynamic;
    function GetNamePath: string; override;
    function HasParent: Boolean; dynamic;
    procedure InsertComponent (AComponent: TComponent);
    procedure RemoveComponent (AComponent: TComponent);
    procedure SetSubComponent (IsSubComponent: Boolean);
    function SafeCallException (ExceptObject: TObject;
        ExceptAddr: Pointer): HRESULT; override;
    function UpdateAction (Action: TBasicAction): Boolean; dynamic;
    function IsImplementorOf (const I: IInterface): Boolean;
    function ReferenceInterface (const I: IInterface;
        Operation: TOperation): Boolean;

    property ComObject: IUnknown read GetComObject;
    property Components [Index: Integer]: TComponent read GetComponent;
    property ComponentCount: Integer read GetComponentCount;
    property ComponentIndex: Integer read GetComponentIndex
        write SetComponentIndex;
    property ComponentState: TComponentState read FComponentState;
    property ComponentStyle: TComponentStyle read FComponentStyle;
    property DesignInfo: Longint read FDesignInfo write FDesignInfo;
    property Owner: TComponent read FOwner;
    property VCLComObject: Pointer read FVCLComObject write FVCLComObject;

published
    property Name: TComponentName read FName write SetName stored False;
    property Tag: Longint read FTag write FTag default 0;
end;

```

9.4.2 属性

TComponent 类的属性可分为两大类, 即声明为 public 类的属性和声明为 published 类的属性。声明为 published 的属性有:

■ Name: TComponentName

Name 是组件实例的名称, 在一个对象内, 每个组件都有一个惟一的组件名称, 在程序代

码中是以组件的名称来标识组件的。在可视化设计中, Delphi 将自动为添加的每个组件产生唯一的名称, 命名方法是在类名上去除类型标识 T, 并在后面加数字而构成的。如 TButton 类, 组件将命名为 Button1、TButton2 等。对于程序开发而言, 这种命名将影响程序的可读性, 会对日后的代码维护造成混乱。一个有良好编程修养的程序员决不会在程序中直接采用这种命名的, 一般的命名方式是将组件的类型与具体相关的操作或所要完成的功能结合在一起, 使程序具有很好的可读性(程序的语义直接明了)。如打开文件的按键可命名为 BtnOpen、BtnOpenFile、BtnOpenFileForReadData 等。在上述的几个命名中, 最后的命名最长, 但可读性是最好的。因此从项目研发和管理的角度来讲, 这种命名风格是最好的, 因为命名的本身就是程序的注释。

■ Tag: Longint

Tag 属性对组件自身而言并无具体的意义与作用, 它是一个整数, 可由组件使用者随意指定以表达一定的意义, 且对组件本身不会造成任何影响。一般编程中是为了区别组件而使用该属性。例如, 假如界面上有三个 TButton 按键分别用于控制图形的放大、缩小和平移操作, 为了使三个组件用同一个 OnClick 事件处理函数, 则可分别对三个按键的 Tag 属性指定 1、2、3 以示区别, 在事件处理函数中可写如下代码来进行不同的处理:

```
TForm1.OnButtonClick(Sender: TObject);
Begin
    Case Sender as TComponent).Tag of
        1: ZoomIn;
        2: ZoomOut;
        3: Move;
    end;
end;
```

在上述代码中若不使用组件的 Tag 属性, 则要用组件名进行区分, 这样代码只能进行字符串比较而不能用 Case 语句进行程序跳转, 在程序的可读性和运行效率方面都将不如上面的代码。

在 TComponent 类中声明在 public 下的组件属性有下列这些:

■ ComObject: IUnknown

该属性仅对支持 COM 接口的对象有意义, 对于支持 COM 接口的组件, 该属性存有组件所实现的接口引用; 应用程序可通过该属性取得接口。对于一般不支持 COM 接口的 VCL 组件, 访问该属性将引发 EComponentError 异常。对于 COM 对象的介绍请参见后面的有关章节的内容。

■ Components[Index: Integer]: TComponent

该属性是一个 TComponent 类的数组, 该数组保存有该组件所拥有的所有从属组件或窗口。实际上每个组件都有一个所有者, 即 Owner 属性, 该属性指向对象所有者, 而 Owner 属性是只读的, 它是在对象创建时由构造函数的 Owner 参数传入的。虽然 Owner 属性是只读的, 但并不意味着组件的所有者是不可更改的, 只是更改不是很容易而已。组件要存放所拥有的对象引用的原因在于: 在可视化设计中, 所有者组件要控制所拥有的组件对象的创建与销毁。对象销毁就是通过将该属性中所有的对象销毁来实现组件的自动销毁功能的。对于 Index 参数, 它是基于 0 开始的索引。在具体的编程中, 通常可通过该数组查找窗口或组件中指定的组件, 例如:

```
procedure TForm1.InvisibileAllButton;  
var  
    i: integer;  
begin  
    for i: = 0 to ComponentCount-1 do  
        begin  
            if Components[i] is TButton then  
                (Components[i] as TButton).Visible: = False;  
            end;  
        end;  
    end;  
end;
```

实际上用该属性来查找组件与用组件名来查找组件是一样的,前者是用索引号来得到一个组件,而 FindComponent 是用组件名来返回组件的。实际上 FindComponent 也是通过查找该属性中的所有组件的组件名来实现的。

■ ComponentCount: Integer

该属性是一个只读属性,其返回组件所拥有的组件个数。它的取值是基于 1 开始的,当属性为 0 时表示组件不拥有组件。

■ ComponentIndex: Integer

用于指示当前所操作的组件在 Components 数组中的索引。

■ ComponentState: TComponentState

该属性用于指示组件当前所处的状态。组件的状态有以下几种:

- csAncestor: 该状态表示这一组件是在父窗口类中引进的组件,只有在 csDesigning 被设置的前提下才能设为该状态,也就是说只有处于设计期的组件才有可能被设为该标识。假设在一个 TForm 窗口中设置了一个 TButton 按钮,然后又设计了一个从该窗口继承下来的窗口类,则在这个新窗口中,继承而得的这个按钮在设计期将被 Delphi 设为 csAncestor 状态。
- csDesigning: 被设置为这一状态的组件表明该组件正处于设计期中,即组件是由窗口设计器所操纵着在进行窗口的可视化设计。在 Delphi 的可视化设计中,设计时放在窗口中的组件并不是 Delphi 根据组件在 DFM 文件中的属性而画在窗口上的,它实际上是一个真实的 Window 窗体,只不过是由 Delphi 的窗体设计器所控制而已,而且组件的构造函数也已运行过。所以,csDesigning 属性在组件设计中是最常用的状态之一。在组件的属性设置函数或组件的构造函数中,常通过该属性来判断组件是否处于设计期而运行不同的程序代码。例如常在程序中写出类似如下的代码:

```
procedure SetDelayTimer (const Delay: integer);  
begin  
    FDelayTime: = Delay;  
    if not (csDesigning in ComponentState) then  
        begin  
            Timer: = TTimer.create (self);  
            Timer.Interval: = 1000;  
            Timer.OnTimer: = HandleTimer;  
            Timer.Enabled: = True;  
        end;  
    end;  
end;
```

- **csDestroying**: 该状态表示组件马上要被销毁。
- **csFixups**: 处于该状态的组件表明组件与另一个还未调入的窗口有关联, 当相关的窗口调入后, 该标识将会被清除。
- **csFreeNotification**: 该状态标识表示组件在销毁时需要向一个或多个组件发出销毁的通知。只有另一个组件调用了该组件的 **FreeNotification** 方法后组件才会被设置为该属性。
- **csInline**: 该标识主要是在窗框 **Frame** 保存和调入时用于区分嵌套的 **Frame** 对象。被设置了该标识的组件是处于窗体最上层的组件, 可以在设计期进行可视化的属性设计或修改。
- **csLoading**: 该标识表示组件正处于装载过程中, 即组件正被一个 **TFile** 或其派生类对象所装载, 只有组件所拥有的所有子组件被装载后该标识才会被清除。
- **csReading**: 当组件被装载时, 组件正由一个流中读取组件属性时被设为此标识, 它往往与 **csLoading** 标识一同被设置, 因为 **csReading** 标识是在组件被装载而还未读入属性的间隔时刻被设置为此标识的。
- **csUpdating**: 该标识表示组件正处于更新过程中以反映出祖先窗口的变化, 只有被设置了 **csAncestor** 标识的组件才有可能设置有该属性。
- **csWriting**: 该标识表示组件正在将其属性写到一个流对象中。
- **csDesignInstance**: 该标识往往与 **csDesigning** 标识一同出现。该标识表示该组件是窗口设计器中的根对象。比如在设计一个窗体时, 该窗体就被设置了这一标识。若同时还存在另一个框体, 且该窗体的作用与一般的组件相类似, 则这个与组件相类似的窗体不会被设置为该标识。

■ **ComponentStyle: TComponentStyle**

该属性是一个集合类型, 用于管理组件与流对象或 **Object Inspector** 进行信息交互, 该属性的取值是组件定义的一部分, 是一个只读属性, 它由组件的构造函数设置。在各标识中, 只有 **csSubComponent** 标识较为特殊, 它可通过调用 **SetSubComponent** 方法设置。该属性的各取值如下:

- **csInheritable**: 该标识表示派生类组件是否能继承组件的这一属性设置。对于一个窗体中的所有对象, 只要有一个组件不设有 **csInheritable** 标识, 则该窗口就不可作为窗体类的祖先, 即该窗体不能有派生类。
- **csCheckPropAvail**: 该标识只有 **COM** 对象才有用, 它用于表示组件要检查属性以得到属性的可读取性。对于 **COM** 对象, **Object Inspector** 并不能直接得知组件的属性是否可读取以显示组件的属性值, 只有经过属性的可读取性检查后才能知道哪些属性是可读取和显示的。
- **csSubComponent**: 表示该组件是一个子组件, 即组件是由 **Owner** 属性所指的组件所拥有。对于子组件, 它与窗体上的最上层组件不同, 在 **DFM** 文件中它不与它所在的窗体或数据模块一起存储, 而是将整个组件作为所有者组件的一个 **published** 属性, 组件的 **published** 属性和事件与组件的所有者的 **published** 属性存放在一起。

- **csTransient**: 该标识表示组件是一个临时对象, 组件的属性不需要保存到 DFM 文件中。

■ **DesignInfo: Longint**

该属性是 Delphi 内部使用的属性, Delphi 的窗口设计器通过该属性可取得组件的有关信息。

■ **Owner: TComponent**

它存有组件的所有者对象指针, 比如在组件的 **Components** 属性所指的每个子对象中, 子对象的 **Owner** 属性将指向本组件。

■ **VCLComObject: Pointer**

该属性是 Delphi 内部使用的, 只对 Com 类对象有意义。它指向实现 COM 接口的对象, 一般应用程序不要使用此属性, 若要取得接口, 可通过访问 **ComObject** 属性取得。

9.4.3 方法

TComponent 类是一个基础类, 是 Delphi VCL 组件的核心类之一, 它的方法都是一些通用的基础方法。对于组件设计人员, 很有必要了解它的方法。它的方法可分为私有方法、保护方法和公有方法三大类。对于一般组件的使用人员, 只需要了解它的公有方法即可。但对于组件设计人员, 仅了解公有方法是远远不够的, 还应了解它的受保护的方法才能便于设计自己和组件。对于组件的私有方法, 因为其只能由 **TComponent** 本身所调用, 了解它的意义不大, 所以本书就不进行介绍了。下面我们先介绍组件的公有方法。

■ **Constructor Create (AOwner: TComponent); virtual**

它是组件的构造函数, 也是一个虚函数, 可以由派生类所覆盖。在该函数中, 它主要做了三件事, 即调用基类的构造函数、设置组件的 **ComponentStyle** 属性为 **csInheritable** 和将自己设置到组件的属主中以作为属主组件的子组件 (对于无属主的组件不需要进行相关的设置)。对于第一件事是所有类所应完成的事, 即在每个组件的构造函数中都有如下类似的一句代码:

```
Inherited;  
或 Inherited Create (owner);
```

■ **Destructor Destroy; override**

组件的析构函数, 它首先销毁组件所拥有的子对象, 然后再调用基类的析构函数。对于任何对象来讲, 都应是先销毁本类所声明的资源, 然后才调用基类的析构函数。它与构造函数刚好相反, 构造函数是先调用基类的构造函数, 然后才进行本组件类的初始化设置工作。

■ **procedure BeforeDestruction; override**

该方法是一个虚方法, 它是在 **TObject** 中被声明的。当程序调用组件的析构函数时, 组件将立即调用此方法以销毁组件内的其他组件, 它是通过检查组件的状态中是否有 **csDestroying** 标识来运行不同的代码的; 若有此标识, 则调用 **Destroying** 函数来完成内部组件的销毁。在应用程序中不需要直接或隐含调用该方法, 只需要编写该方法的覆盖函数即可。对于应用程序开发, 一般不会用到该方法; 但若是组件设计, 了解该方法还是有好处的。覆盖函数内部, 首先应调用祖先类的该方法, 然后才是编写本组件的相关代码。假如在组件 **TMyComponent** 中定义了一个 **TFont** 的对象 **FFont**, 且该对象在组件的构造函数中被创建, 则在组件销毁时应先销毁

FFont 以保证资源能被回收。而销毁 FFont 的代码既可在 Destroy 函数中,也可编写在 BeforeDestruction 中,以下两种写法的作用是相同的。

```
Destructor TMyComponent.Destroy;  
Begin  
  If assigned (FFont) then FFont.free;  
Inherited;  
End;  
  
Procedure TMyComponent.BeforeDestruction;  
Begin  
  Inherited;  
  If Assigned (FFont) then FFont.free;  
End;
```

仔细比较上面的代码,有一点不同,即在 Destroy 函数中要先销毁组件内的其他对象,然后才调用祖先类的方法,而在 BeforeDestruction 中却相反。对于 Destroy 中的写法,这是对象的销毁顺序所要求的,对于 BeforeDestruction 方法中的写法,Delphi 要求要这样写,而且这样写也较符合覆盖函数的一般编写习惯;根据对其代码的分析,实际上不按这种顺序编写也是正确的。

■ procedure DestroyComponents

该方法销毁组件所拥有的所有组件。该方法是由组件自动调用的,不需要应用程序调用。实际上是由组件的析构函数所调用的。

■ procedure Destroying

该方法在组件销毁前调用组件所拥有的所有组件的 Destroying 方法,以使所拥有的组件能在销毁前释放组件内的资源,它在组件的析函数 Destroy 的最前面被调用,在组件销毁时,若组件正处在销毁过程中(在 ComponentState 中有 csDestroyings 标识)也会由 BeforeDestroy 函数所调用。

■ function ExecuteAction (Action: TBasicAction): Boolean; dynamic

该方法是动态方法,它主要为有窗口句柄的可视组件而设计。当组件发动某动作时,组件将调用可视组件属性中 Action 属性指定的动作。通过组件对 Action 属性的支持,可大大方便程序的界面设计。

■ function FindComponent (const AName: string): TComponent

从组件所拥有的组件中查找指定组件名的组件,若没找到则返回 nil。

■ procedure FreeNotification (AComponent: TComponent)

将指定的组件加入到组件内部的一个列表中。当本组件要被销毁时,组件将依次向列表中的各个组件发出本组件要被销毁的消息。若组件的所有者与指定组件的所有者相同时,则组件不会被加入到组件列表中,因为组件的所有者会向组件发送该消息。该方法主要是用在两个组件有关联的情况下,当一组件销毁时,需向另一组件发出信号,以使在该组件被销毁后,另一组件也能正常工作。例如,本组件需要用另一个窗口的一个组件来输入组件的属性,当本组件销毁时,需向另一窗口中的属性输入组件发出本组件要销毁的消息,以使属性输入组件不会因本组件的销毁而引发异常。该方法在设计具有复杂关系的多个对象时特别有用。

■ procedure RemoveFreeNotification (AComponent: TComponent)

该方法与前一方法的作用刚好相反。它是将指定的组件从组件内部需要发送销毁消息的列表中除去。应用程序没有必要调用此方法，它主要是为组件内部提供的，是为防止两个组件相互发送组件销毁消息而导致程序循环而编写的。

■ procedure FreeOnRelease

该方法是为支持 COM 接口的组件而设计的。当组件所支持的接口要被释放时，组件将调用此方法来释放接口。它也是一个由内部调用的方法。

■ function GetParentComponent: TComponent; dynamic

该方法取得组件的父组件，它主要是为组件属性的存储而设计的。需要注意的是，组件的所有者 Owner 与组件的父组件是有区别的。大多数情况下组件的所有者与组件的父组件是同一个。实际上，组件的父组件主要负责组件属性的存储，而组件的所有者主要负责组件的销毁。有时组件的所有者与组件的父组件是不同的，例如在程序界面设计中，为了使程序的设计便于分工协作，同时也为提高程序的重用性，常将程序分成多个模块，每个模块有自己的窗口与界面，以便各模块独立调试，最后再通过一个主控模块将各模块组合在一个窗口中。在主控程序界面设计时，将在主界面上按界面的需要设置多个 Pannel 对象，然后在主控程序中动态创建各模块的窗口，最后将模块窗口的父组件设置为主控窗口的某个 Pannel 对象，这样模块窗口就成为主控界面的一块。这样一来，通过对模块窗口属性的合理设置，在主控界面上将感觉不到模块窗口的存在。对于模块窗口而言，它的所有者是主控窗口，但它的父组件却是主控窗口中的某个 Pannel 对象。

■ function HasParent: Boolean; dynamic

返回组件是否有父组件的判断，TRUE 表示有父组件。对于有父组件的组件，它必须覆盖 GetParentComponent 和 SetParentComponent 两个方法。对于非可视组件而言，组件一般不需要有父组件；但对于可视化组件，则是有父组件的组件，因此需要覆盖上述两个方法。

■ procedure InsertComponent (AComponent: TComponent)

该方法将指定的组件加入到 TComponent 组件所拥有的组件列表 FComponents 中。需要注意的是，AComponent 所传入的组件必须没有指定组件名 (Name 属性) 或组件名在组件所拥有的组件中是惟一的。在可视化设计中，当我们向窗口添加组件时，可视化窗口设计器就是通过调用此方法来向 TForm 中加入组件的。

■ procedure RemoveComponent (AComponent: TComponent)

该方法从组件所拥有的组件列表中删除一指定的组件。

■ procedure SetSubComponent (IsSubComponent: Boolean)

该方法返回本组件是否是一个子组件的判断。所谓的子组件就是组件的所有者既不是窗口对象也不是数据模块对象的组件。对于子组件，组件的 published 属性将不写到窗体文件中。该方法只在设计期被 Delphi 所调用。

■ function UpdateAction (Action: TBasicAction): Boolean; dynamic

更改组件相关联的 Action 对象。

■ function IsImplementorOf (const I: IInterface): Boolean

返回组件是否支持指定接口的判断。该方法的作用与接口中的 QueryInterface 方法相同。

■ function ReferenceInterface (const I: IInterface; Operation: TOperation): Boolean

该方法建立或断开与指定接口的联系。当在一个组件的属性中有属性接口的属性时, 可以通过该方法建立或断开与指定接口的联系。若有联系时, 当接口被释放, 接口将向组件发出释放的通知。参数 I 是一接口, Operation 用于指定是建立联系还是断开联系, 当 Operation 为 opInsert 时建立联系, 而为 opRemove 时断开联系。

关于 TComponent 组件的私有方法和保护类方法在此就不一一介绍了, 有兴趣的读者可参看 Delphi 的在线帮助文档。对于组件设计人员, 多了解一些底层组件的架构和 VCL 的实现机理有助于设计出复杂而功能强大的组件。

9.4.4 组件的从属关系

TComponent 类引入了 VCL 中组件从属关系的重要概念。有两个属性 Owner 和 Components 支持这种拥有和被拥有概念。每一个组件及其派生类都有 Owner 的属性, 该属性返回组件的属主 (Owner)。类似的一个组件可以拥有其他的组件, 这些组件则成为其属主的 Components 属性中的一个元素。一个组件的构造方法通过调用单变量参数来指明所创建的组件的基类。而属性 Components 最为重要的一个特性则是使得一个组件能够自动销毁其所拥有的所有组件。这点对于程序的设计者十分重要。只要一个组件有自己的属主, 那么当该属主被销毁时组件将会自动被销毁掉。例如 TForm 是 TComponent 的一个派生类, 因此当一个 Form 被销毁时, Form 上的所有组件都将会被销毁, 当然这是在假定所有组件的销毁方法都被正常地调用成功的基础上得到的。下面让我们再来介绍一下 TComponent 的一些方法。最为重要的方法之一便是 Notification。无论一个组件从其属主的 TComponent 组件列表中插入或删除, 都会触发 Notification 事件。举个具体的例子来说明这一点。TForm 是 TComponent 的派生类, 因此具有 Components 的属性, 那么当你加入一个新的组件到 Form 上时, 该组件将加入到该 Form 的 Components 组件列表当中。同时该 Form 还要通过调用 Notification 事件来通知该列表中的其他组件。这个事件在设计时尤其重要, 如图 9-5 所示, 在窗体 Form7 中拥有两个组件 Label1 和 Edit1, Label1 通过属性 FocusControl 引用 Edit1 组件, 那么现在假设删除 Edit1 组件, 如果 Label1 未接到此变化的通知, 那么此时再次选中 Label1, 由于 Edit1 的删除, 此时再访问 Label1 的属性 FocusControl 的值, 将会产生一个冲突。幸运的是 TLabel 组件具有 Notification 方法来避免出现上述问题。

无论何时从 Form 上插入或移走任一个组件都会触发 Notification 方法, 正如下面标签控件 (TCustomLabel) 的 Notification 方法实现代码所示。通过 FocusControl 属性判断被 Label 引用的组件是否被移走 (FfocusControl 是 FocusControl 的内部引用的私有数据), 如果被移走的话, 该引用属性将赋为 nil, 从而避免存取冲突。

```
procedure TCustomLabel.Notification ( AComponent: TComponent;  
Operation: TOperation );  
begin  
    inherited Notification ( AComponent, Operation );  
    if ( Operation = opRemove ) and ( AComponent = FFocusControl )
```



```

then
    FFocusControl := nil;
end;

```

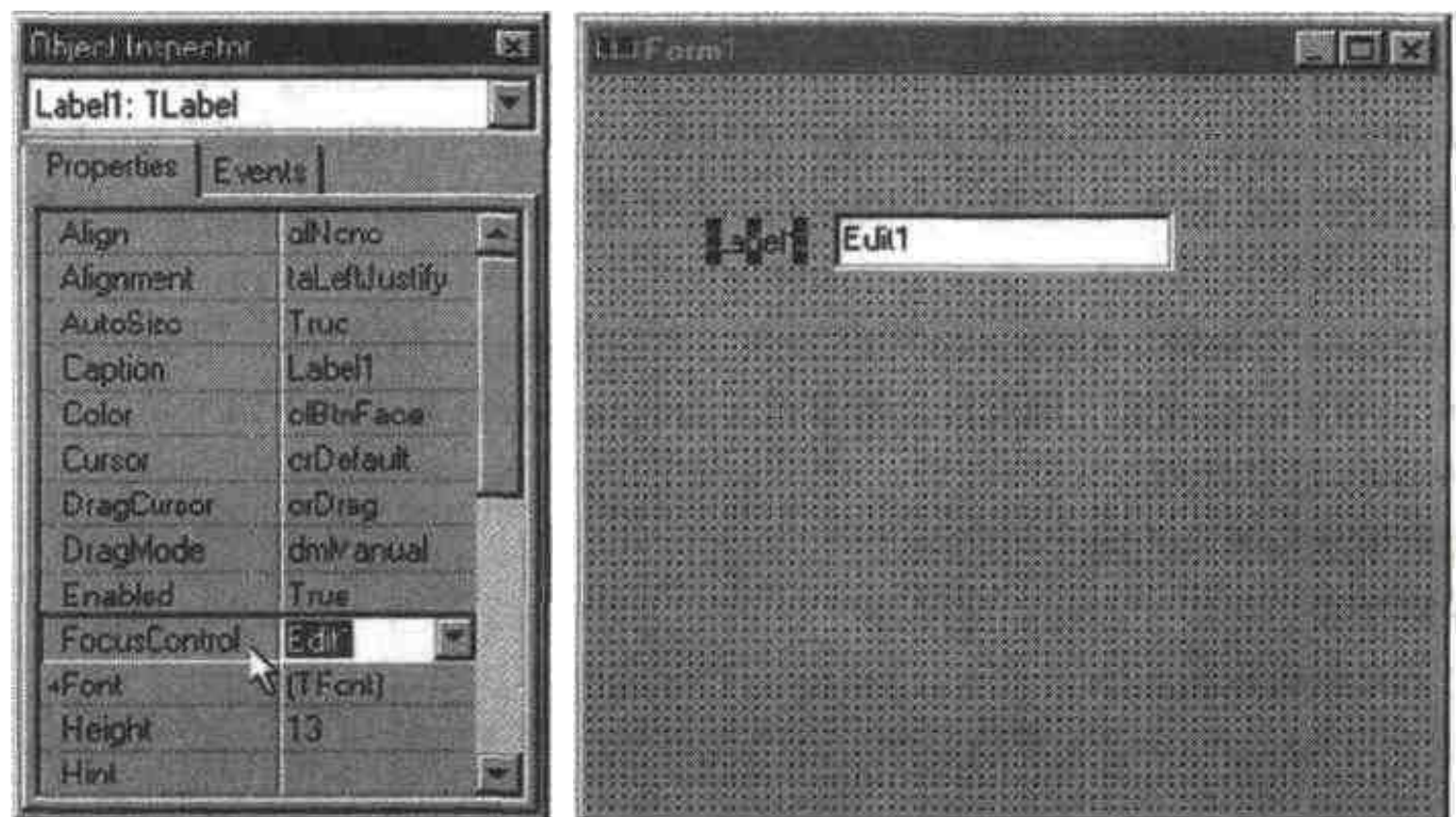


图 9-5 通过 FocusControl 属性连接 Edit 到一个 Label 上

Notification 模式使得处在同一个 Form 中的组件可以顺利工作。但 Delphi 从 2.0 版本后引入了窗体链接的概念。它允许一个组件引用另一个 Form 中的组件，而 Notification 只对处于同一个组件对象列表中的组件起作用。不过不要担心，因为 Delphi 进一步延伸了 Notification 的概念，这就是 FFreeNotifies。当一个组件销毁时，它调用的 Destruct 方法将会修改 FFreeNotifies，同时调用组件列表中每个组件的 Notification 方法。那么一个组件如何获取其他组件的 FFreeNotifies 事件呢？这主要是通过使用新的 FreeNotification 方法来实现的。下面的代码通过 TCustomLabel 对象，解释了具体实现的方法：

```

procedure TCustomLabel.SetFocusControl ( Value: TWinControl );
begin
    FFocusControl := Value;
    if Value < > nil then
        Value.FreeNotification ( Self );
end;

```

当一个新组件赋值给 FocusControl 属性时将会触发 SetFocusControl 方法。只要该组件不为空，所引用的组件的 FreeNotification 方法就将被调用。通过传递 Self 参数给 FreeNotification，被引用的组件 FFreeNotifies 表中将会包含当前的 Label 实例。这样如果引用组件被删除，则该 Label 的 Notification 事件将会被调用。即使该组件在不同的窗体上也会如此。

9.5 TApplication: 应用程序对象

9.5.1 概述

TApplication 是 Delphi 应用程序的类型，在单元 Forms 中声明。该类从 TComponent 继承，是

TComponent 的直接派生类。每个传统的 Delphi 应用程序都封装在一个 TApplication 对象中。该对象包含了程序的主窗体的句柄，通过该句柄，Windows 操作系统可以向应用程序发送消息。实际上我们可以把 TApplication 看成是一个运行时不显示的窗体。

在 Forms 中有个公用全局对象 Application，其方法和属性集中包括了 Windows 操作系统中创建、运行和销毁应用程序等既定的基本操作和属性，因此在用 Delphi 编写 Windows 应用程序时，可以直接使用全局对象变量 Application，这样就简化了用户和 Windows 环境之间的接口。

除了 Windows 句柄外，Application 对象还包含对应用程序的主窗体、帮助文件、应用程序标题的引用，并可以接收到应用程序层的事件。TApplication 封装了以下四个功能：

- Windows 消息处理。
- 菜单加速和键盘处理。
- 异常处理。
- 上下文联机帮助。

下面介绍 TApplication 中的常用属性、方法和事件。

9.5.2 属性

■ Active

属性 Active 指明了应用程序是否处于活动状态且拥有焦点。定义如下：

```
property Active: Boolean;
```

Active 是只读属性。当应用程序是活动状态时 Active 为 True，否则为 False。TApplication 的构造函数置 Active 为 True。如果窗口或应用程序拥有焦点，那么该应用程序是活动的。当其他应用程序的窗口成为活动的时，当前应用程序即为非活动的。应用程序关闭时 TApplication 的析构函数置 Active 为 False。

在应用程序中，可以用一个计时器 (TTimer) 来检查属性 Active 的值，从而确定当前的应用程序是否是活动状态，以便做出相应的处理。也可以在事件 OnActive 和事件 OnDeactive 中定义指定的操作。

■ DialogHandle

属性 DialogHandle 提供使 Delphi 应用程序使用非 Delphi 对话框的一种机制。定义如下：

```
property DialogHandle: Hwnd;
```

当使用 API 函数 CreateDialog 创建一个非模式化对话框时需要使用 DialogHandle，并且需要查看应用程序消息循环中的消息以进行相应的操作。比如，当一个非模式化对话框收到一条激活消息 (WM_NCACTIVATE) 时可以将其句柄赋值到 DialogHandle，当对话框收到一条解除激活消息时置 DialogHandle 为 0。

■ 属性 ExeName

属性 ExeName 包含了可执行的应用程序文件名及其路径信息。定义如下：

```
property ExeName: string;
```

ExeName 是只读属性。使用 ExeName 能够得到应用程序可执行文件的文件名。

这是一个很有用的属性。例如, 运行光盘中的应用程序 f: \ media \ myapp.exe 时, 可能需要访问目录 f: \ media \ data \ 中的文件或者确定应用程序所在的驱动器盘符。这时可以使用 Delphi 提供的函数 ExtractFilePath 和 ExtractFileName 对属性 ExeName 进行解析, 从而得到需要的信息。下面是一个常见的复制文件的例子:

```
procedure TForm1.Save1Click (Sender: TObject);

var
  NewFileName: string;
  Msg: string;
  NewFile: TFileStream;
  OldFile: TFileStream;
begin
  NewFileName := ExtractFilePath (Application.ExeName)
                + ExtractFileName (Edit1.Text);
  Msg := Format (' Copy %s to %s? ', [Edit1.Text, NewFileName]);
  if MessageDlg (Msg, mtCustom, mbOKCancel, 0) = mrOK then
  begin
    OldFile := TFileStream.Create (Edit1.Text, fmOpenRead
                                   or fmShareDenyWrite);
    try
      NewFile := TFileStream.Create (NewFileName, fmCreate
                                     or fmShareDenyRead);
      try
        NewFile.CopyFrom (OldFile, OldFile.Size);
      finally
        FreeAndNil (NewFile);
      end;
    finally
      FreeAndNil (OldFile);
    end;
  end;
end;
```

■ 属性 Handle

属性 Handle 提供了对应用程序主窗口句柄的访问。定义如下:

```
property Handle: HWND;
```

当调用一个需要父窗口句柄的 Windows API 函数时需要使用 Handle 属性。例如, 应用程序中某个动态链接库 (DLL) 可能需要父窗口句柄以使得其自身能够弹出并且显示在最前端。使用 Application.Handle 构成应用程序的若干窗口, 使得这些窗口在应用程序中能够被最小化、恢复、有效或无效。

注意 编写一个使用 VCL 窗体的动态链接库时, 应将主运行程序中主窗口的句柄赋值到该动态链接库的 Application.Handle 属性。这样就使得动态链接库的窗体成为主应用程序的一部分。需要特别指出的是, 永远不要在 EXE 应用程序中给 Application.Handle 赋值。

■ 属性 HelpFile

属性 HelpFile 指明了应用程序用于显示帮助内容的文件名。定义如下：

```
property HelpFile: string;
```

使用 HelpFile 是为了应用程序拥有一个使用标准 Windows 帮助系统的帮助文件。Windows 显示由 HelpFile 属性指明的帮助文件。要让应用程序实现这一点，必须在运行时为 HelpFile 属性赋予一个文件名的值，或者在设计时 Project Options 对话框的 Application 页面中指定一个帮助文件，如图 9-6 所示。在默认情况下，HelpFile 是一个空串 (""），并且应用程序的帮助方法忽略所有的试图显示帮助。如果 HelpFile 包括任何内容，帮助主题的方法将根据文件名调出 Windows 帮助系统以提供联机帮助。

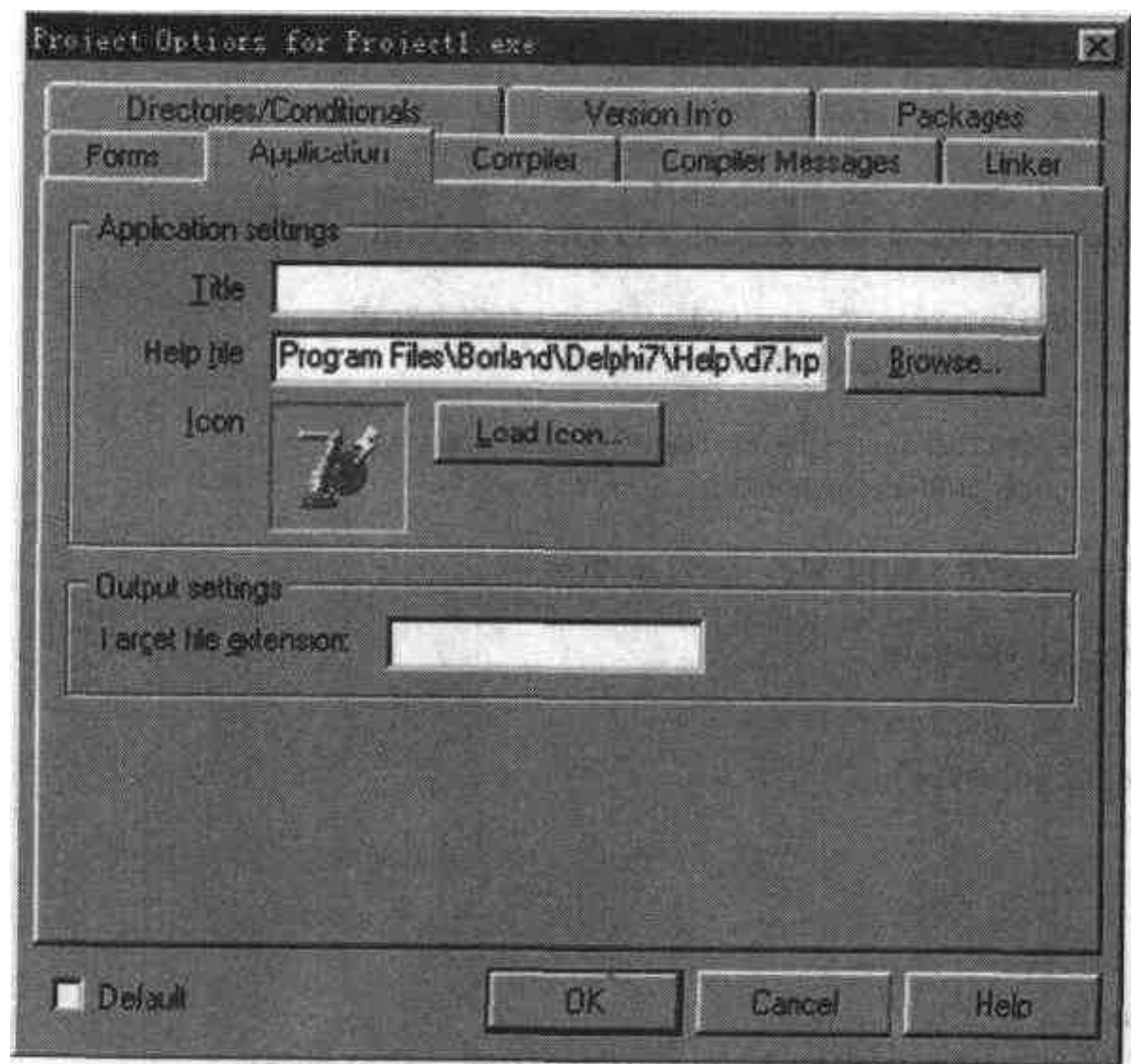


图 9-6 通过 Project Options 对话框的 Application 页面指定一个帮助文件

注意 如果活动窗口的帮助文件已指定，则该文件将优先于应用程序的帮助文件显示。

■ 属性 Hint

属性 Hint 指明了出现在帮助提示框 (help hint box) 中的文本字符串。定义如下：

```
property Hint: string;
```

TApplication.Hint 属性就是鼠标正在移动经过的控件或菜单项的 Hint 值。该属性也能被赋予一个向用户提供动作、错误或其他信息的字符串值。因此，使用 Hint 属性也能够从控件传递提示信息到另一显示区。例如通过 OnHint 事件句柄将提示显示到状态栏中，这时是读取 Hint

属性；当应用程序正在进行某一动作时简要描述其状态，这时是设置 Hint 属性。当 OnHint 事件发生时帮助提示才出现。因此，即使 TApplication 的 Hint 属性被赋予状态栏的标题（例如状态栏控件的标题显示 Hint 属性的当前字符串值），也应归于 OnHint 事件。

Hint 字符串包括两部分：短提示和长提示。短提示用于简洁的弹出提示；长提示与短提示之间用竖线“|”隔开，用于在状态栏中显示较详细的提示信息。可以使用单元 Controls 中提供的字符串函数 GetShortHint 和 GetLongHint 从 Hint 属性中分解得到短提示和长提示。

注意 当应用程序通过设置 Hint 属性向用户提供某一事件发生的信息时应当切记，在默认情况下，当鼠标移动经过某一控件时，Hint 字符串被复位到该控件的 Hint 属性值。

■ HintHidePause

属性 HintHidePause 指定了鼠标尚未从控件或菜单项上移开、在隐藏帮助提示之前的时间间隔。定义如下：

```
property HintHidePause: Integer;
```

用 HintHidePause 以毫秒为单位指定等待时间，在 TApplication 的构造器中该属性被置为 2500 毫秒（2.5 秒）。控件或菜单项的帮助提示在 Hint 属性中指定。

注意 默认应将 HintHidePause 的值预先确定为 HintPause 属性值的 3~5 倍较合适。

■ MainForm

属性 MainForm 惟一标识了应用程序的主窗体。定义如下：

```
property MainForm: TForm;
```

属性 MainForm 指定的窗体不一定等同于应用程序的主窗口。属性 MainForm 的值一定是由方法 CreateForm 创建的所有窗体中的第一个窗体，但该窗体未必是应用程序的主窗口。新建一个项目时，MainForm 属性值自动置为 Form1。在设计时可通过 Project Options 对话框中的 Forms 页面在多个窗体中指定其中之一为 MainForm。在运行时是不能修改 MainForm 属性的，因为该属性是只读的。主窗体是应用程序主体创建的第一个窗体。主窗体关闭即应用程序终止。在应用程序中，可以调用方法 Application.MainForm.Close 来终止应用程序运行，并可以获取 Application.MainForm.Top、Application.MainForm.Left 等属性的值从而确定当前活动窗口的位置以及尺寸等。

■ ShowMainForm

属性 ShowMainForm 确定了是否在应用程序启动时显示主窗体。定义如下：

```
property ShowMainForm: Boolean;
```

应用程序用 ShowMainForm 属性控制是否以及何时显示其主窗体。TApplication 的构造函数置 ShowMainForm 为 True。默认情况下主窗体将被显示，并在 MainForm 属性中指明了主窗体。

如果需要在应用程序启动时隐藏主窗体，那么应在主项目文件中调用 Application.Run 之前将 ShowMainForm 设为 False，并且确信主窗体的 Visible 属性值为 False。在许多实现 OLE 自动化服务器中，这是很有用的，比如在启动自动化服务时隐藏服务器程序的主窗体。

另外,如果需要在应用程序启动时显示一个闪屏(Flash)并为应用程序的环境做一些准备,同时需要禁止主窗体显示,这也可以利用属性 ShowMainForm 来实现。比如 Delphi 启动时就禁止了主窗体的显示。

■ 属性 Terminated

属性 Terminated 报告程序是否收到终止程序的 Windows 消息 WM_QUIT。定义如下:

```
property Terminated: Boolean;
```

Terminated 是只读属性。该属性主要用于调用 ProcessMessages 方法时应用程序不必在停止后试图处理 Windows 消息。当 ProcessMessages 方法收到消息 WM_QUIT 时, Terminated 将被置为 True。

Delphi 应用程序总会因为主窗体或应用程序关闭,或者因为 Terminate 方法被调用而收到消息 WM_QUIT。

当应用程序执行强度较大、占用系统资源较多的运算时,应当周期性地调用 Application.ProcessMessages 方法,并检查属性 Application.Terminated 以确定是否需要终止运算从而终止应用程序。

■ UpdateFormatSettings

属性 UpdateFormatSettings 指明了当用户改变系统配置时应用程序是否自动更新格式设置。定义如下:

```
property UpdateFormatSettings: Boolean;
```

使用 UpdateFormatSettings 属性,应用程序可以控制自动更新格式设置。TApplication 的构造函数置该属性为 True。当应用程序收到消息 WM_WININICHANGE 时将检查 UpdateFormatSettings 属性。建议使用默认的格式设置,也就是 Windows 本地的设置。可以置 UpdateFormatSettings 为 False 以避免在 Delphi 应用程序执行期间改变格式设置。

■ UpdateMetricSettings

属性 UpdateMetricSettings 属性指明是否对提示窗口字体和图标标题等相关设置进行更新。定义如下:

```
property UpdateMetricSettings: Boolean;
```

UpdateMetricSettings 属性指明了系统中提示窗口字体及图标、标题等设置的改变是否反映到应用程序中相关设置的改变。TApplication 的构造器置 UpdateMetricSettings 的初始值为 True。

9.5.3 方法

■ BringToFront

方法 BringToFront 设置应用程序中最近一次的活动窗口到桌面上所有窗口的最前端。其定义如下:

```
procedure BringToFront;
```

用 BringToFront 方法可以找到属于主窗体的最近一次的活动窗口并且将其置于最前端。BringToFront 方法也可以测试和查看一个窗口在成为最前端窗口之前是否是可见的(Visible)和

有效的 (Enabled)。例如, 当应用程序收到邮件时, 可能需要将专门的处理程序激活并置于 Windows 桌面的最前端。这时就可以调用 `Application.BringToFront` 方法来实现。

■ CreateForm

方法 `CreateForm` 方法用于创建新的窗体。定义如下:

```
procedure CreateForm (FormClass: TFormClass; var Reference);
```

Delphi 应用程序总会调用 `CreateForm` 方法。因此程序员很少有必要直接调用 `CreateForm` 方法。一个典型的 Delphi 项目在项目的主体代码部分包括一处或多处调用 `CreateForm` 方法, 并且在使用窗体设计器时自动控制窗体的创建。也可以在运行时调用 `CreateForm` 方法来动态创建窗体。`CreateForm` 方法根据 `FormClass` 参数创建一个新的指定的窗体并且将窗体赋予到变量参数 `Reference`。新创建的窗体的所有者就是对象 `Application`。应用程序将第一个调用 `CreateForm` 创建的窗体默认为项目的主窗体。

■ HandleException

方法 `HandleException` 为应用程序的异常提供默认的句柄。定义如下:

```
procedure HandleException (Sender: TObject);
```

方法 `HandleException` 对于编写特定组件的作者来说是有用的, 因为它可以产生一个不必对 Windows 消息产生响应的事件。在应用程序中可以利用 `OnException` 事件句柄将其他的异常操作控制在自定义的代码中。在应用程序代码中, 如果异常跳过了所有的 `try` 块, 那么应用程序将自动调用 `HandleException` 方法, 并将显示一个提示有错误发生的对话框。除非异常对象是 `EAbort`, 此时 `HandleException` 将调用 `OnException` 句柄 (如果存在); 否则将调用 `ShowException` 显示一个提示有错误发生的对话框。

■ UnhookMainWindow

`UnhookMainWindow` 方法用于销毁由 `HookMainWindow` 方法挂在主窗体的程序。定义如下:

```
type TWindowHook = function (var Message: TMessage): Boolean of object;  
procedure UnhookMainWindow (Hook: TWindowHook);
```

用 `UnhookMainWindow` 可以销毁挂钩窗口。在参数 `Hook` 中指明对话框过程。`TWindowHook` 类型是调用 `HookMainWindow` 方法的参数。该参数是非 Delphi 对话框中调用对话框程序的方法指针。对话框程序与窗口程序相似, 都是为对话框处理消息, 只是语法不同。

9.5.4 事件

■ OnActivate

当应用程序成为活动状态时 `OnActivate` 事件发生。定义如下:

```
type TNotifyEvent = procedure (Sender: TObject) of object;  
property OnActivate: TNotifyEvent;
```

用 `OnActive` 事件编写一个事件句柄来完成当应用程序成为活动状态时指定特别的处理。当一个 Windows 应用程序最初运行时或其焦点从另一个 Windows 应用程序转移回到当前应用程序时, 该应用程序就成为活动状态。

■ OnDeactivate

当应用程序成为非活动状态时 OnDeactivate 事件发生。定义如下：

```
type TNotifyEvent = procedure (Sender: TObject) of object;  
property OnDeactivate: TNotifyEvent;
```

在应用程序成为非活动状态之前可以立即触发 OnDeactive 事件，从而完成在该事件句柄中指定的特别处理。当用户从当前应用程序转换到另一应用程序时，当前应用程序的 OnDeactive 事件即发生。

■ OnException

当应用程序中的某个无句柄的异常发生时就触发了事件 OnException。定义如下：

```
type TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;  
property OnException: TExceptionEvent;
```

可以通过 OnException 事件来改变在应用程序中无句柄的异常发生时的默认动作。在方法 TApplication.HandleException 方法中，OnException 事件句柄被自动调用。

OnException 事件仅用于处理在进行消息处理时发生的异常。在 Application.Run 执行前或执行后发生的异常不会导致 OnException 事件发生。

如果某个异常在应用程序代码的 try 块中被忽略，那么应用程序将自动调用 HandleException 方法。除非异常对象是 EAbort，此时 HandleException 将调用 OnException 句柄（如果存在）；否则将调用 ShowException 显示一个提示有错误发生的对话框。TExceptionEvent 类型是 OnException 事件的类型，该类型在应用程序中指向一个处理异常的方法。参数 Sender 是引发异常的对象，而参数 E 是异常对象。

■ OnHelp

当应用程序收到帮助请求时 OnHelp 事件发生。定义如下：

```
type THelpEvent = function (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean of object;  
property OnHelp: THelpEvent;
```

用 OnHelp 编写一个事件句柄以完成有请求帮助时特别的处理。HelpContext 方法和 HelpJump 方法自动引发 OnHelp 事件。在事件发生之后置 CallHelp 为 True 以使 VCL 调用 WinHelp；置 CallHelp 为 False 以防止 VCL 调用 WinHelp。Delphi 应用程序中所有与帮助有关的方法都经过 OnHelp 事件。仅当 OnHelp 事件中的 CallHelp 参数返回 True 或 OnHelp 事件没有被指定到有效的句柄时，WinHelp 被调用。

■ OnHint

当鼠标指针移动经过某个控件或菜单项并且该控件或菜单项能够显示帮助提示时，事件 OnHint 发生。定义如下：

```
type TNotifyEvent = procedure (Sender: TObject) of object;  
property OnHint: TNotifyEvent;
```

用 OnHint 编写的事件句柄能够在 OnHint 事件发生时执行指定的操作。当用户停放鼠标指针在某个控件上，并且该控件的 Hint 属性值不是空串 ('') 时，OnHint 事件将发生。通常用 OnHint 事件显示控件或菜单项 Hint 属性的值作为某个面板控件（如 TStatusBar）的标题，因此

把面板 (panel) 用做状态栏 (status bar)。当 OnHint 事件发生时, Hint 属性通常被指定为一个帮助提示 (help hint) 和一个在别处显示的长提示 (longer hint)。

■ OnIdle

当应用程序成为空闲状态时 OnIdle 事件发生。定义如下:

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;  
property OnIdle: TIdleEvent
```

用 OnIdle 编写一个事件句柄, 当应用程序空闲时完成指定的操作。当应用程序不执行任何代码时即为空闲的。例如, 当应用程序等待用户输入时该应用程序是空闲的。TIdleEvent 类型是 OnIdle 事件的类型, 它指向一个当应用程序空闲时运行的方法。对象 TIdleEvent 有一个默认为 True 的布尔变量 Done。当 Done 为 True 时, Windows API 函数 WaitMessage 将在 OnIdle 返回时被调用。WaitMessage 使其他应用程序得到控制焦点直到应用程序的消息队列中出现一条新的消息。当参数 Done 为 False 且应用程序不忙时, 应用程序不会使其他应用程序得到控制焦点。当应用程序转为空闲状态时, OnIdle 事件仅发生一次, 直到参数 Done 置为 True 才可能发生下一次 OnIdle 事件。应用程序置 Done 为 False 消除了紊乱的 CPU 时间计数, 而该计数可能影响整个系统的性能。

■ OnMessage

当应用程序接收到 Windows 消息时事件 OnMessage 发生, 定义如下:

```
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of object;  
property OnMessage: TMessageEvent
```

用于接收 Windows 消息, 该事件能接收程序向 Windows 发送的所有消息。应用程序接收到一个消息时产生该事件。变量 Msg 是 Windows 消息类型。

第 10 章 深入浅出 VCL (下)

10.1 TThread: 线程对象

10.1.1 概述

线程是操作系统对象，描述代码在特殊进程中执行的路径。每个 Win32 应用程序至少有一个线程，我们称之为主线程或缺省线程，不过应用程序中可以创建其他线程来执行其他任务。线程提供了同时运行多个特定代码过程的手段。线程给 Windows 开发人员带来了很多好处，使我们可以应用程序运行的同时，创建一些用于后台处理的线程。

大多数 VCL 都假设在任何给定时间内只能有一个线程访问它。这种限制在 VCL 中与用户界面相关的组件中表现得尤为明显，例如控件（从 TControl 类继承下来的组件）都不是线程安全的。甚至许多非用户界面 VCL 组件也都不是线程安全的，比如 TList 就不是为多线程同时操作而设计的。如果需要使用多线程的话，必须使用 TThreadList 来代替 TList。值得一提的是 VCL 整个 Graphics 单元中的绘图对象（如：TCanvas、TPen、TFont、TIcon 等）是线程安全的，VCL 的流机制也是线程安全的，它们保证这些类可以被多个线程同时有效地读写。

TThread 是 Delphi 把 API 中的线程封装成的一个 Object Pascal 对象。将线程作为类来封装有着许多优点。首先它能清晰、安全地界定线程相关的局部变量和进程相关的全局变量。类—对象的模型到实体的映射关系保证了声明在类中的任何变量都是局部的，声明在类外的任何变量都是全局的。所以在写新线程的 Execute 函数时只要注意对类外部的变量、方法的访问就可以了，至于类内部的变量、方法则可以任意使用而不用考虑同步的问题。将线程封装成类的重要的好处是写新线程的时候可以充分利用类的优点。可以通过继承来重用父类的功能，这实在是一个激动人心的功能。

10.1.2 线程对象的封装和运行机制

既然已经知道将线程封装成类有诸多好处，那么，作为一个 Delphi 程序员了解 Delphi 是如何将线程封装成类以及如何使用 TThread 对象将对开发多线程程序大有帮助。

Delphi 7 中 TThread 类是这样声明的：

```
TThread = class
private
{$IFDEF MSWINDOWS}
    FHandle: THandle;
    FThreadID: THandle;
{$ENDIF}
{$IFDEF LINUX}
    // * * FThreadID is not THandle in Linux * *
    FThreadID: Cardinal;
    FCreateSuspendedSem: TSemaphore;
    FInitialSuspendDone: Boolean;
```

```

{$ENDIF}
    FCreateSuspended: Boolean;
    FTerminated: Boolean;
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FSynchronize: TSynchronizeRecord;
    FFatalException: TObject;
    procedure CallOnTerminate;
    class procedure Synchronize (ASyncRec: PSynchronizeRecord); overload;
{$IFDEF MSWINDOWS}
    function GetPriority: TThreadPriority;
    procedure SetPriority (Value: TThreadPriority);
{$ENDIF}
{$IFDEF LINUX}
    // * * Priority is an Integer value in Linux
    function GetPriority: Integer;
    procedure SetPriority (Value: Integer);
    function GetPolicy: Integer;
    procedure SetPolicy (Value: Integer);
{$ENDIF}
    procedure SetSuspended (Value: Boolean);
protected
    procedure CheckThreadError (ErrCode: Integer); overload;
    procedure CheckThreadError (Success: Boolean); overload;
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize (Method: TThreadMethod); overload;
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
public
    constructor Create (CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure AfterConstruction; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: LongWord;
    class procedure Synchronize (AThread: TThread; AMethod: TThreadMethod); overload;
    class procedure StaticSynchronize (AThread: TThread; AMethod: TThreadMethod);
    property FatalException: TObject read FFatalException;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
{$IFDEF MSWINDOWS}
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
{$ENDIF}
{$IFDEF LINUX}
    // * * Priority is an Integer * *
    property Priority: Integer read GetPriority write SetPriority;
    property Policy: Integer read GetPolicy write SetPolicy;
{$ENDIF}
    property Suspended: Boolean read FSuspended write SetSuspended;

```

```

{$IFDEF MSWINDOWS}
    property ThreadID: THandle read FThreadID;
{$ENDIF}
{$IFDEF LINUX}
    // * * ThreadID is Cardinal * *
    property ThreadID: Cardinal read FThreadID;
{$ENDIF}
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
end;

```

从声明中可以看出, TThread 是直接继承于 TObject 类, 因此它不是组件。由其中的 {\$IFDEF LINUX} 可以看出 TThread 是设计成兼容 Delphi 和 Kylix 的, 满足了跨平台的需要。需要注意 TThread.Execute 方法是抽象方法, 这意味着 TThread 类是抽象类, 因此无法直接创建 TThread 的实例。我们只能创建 TThread 派生类的实例。创建 TThread 派生类实例最简单的方法是在 Delphi 中选择 File|New 菜单, 然后在 New Items 对话框中选择 Thread Object, 如图 10-1 所示。

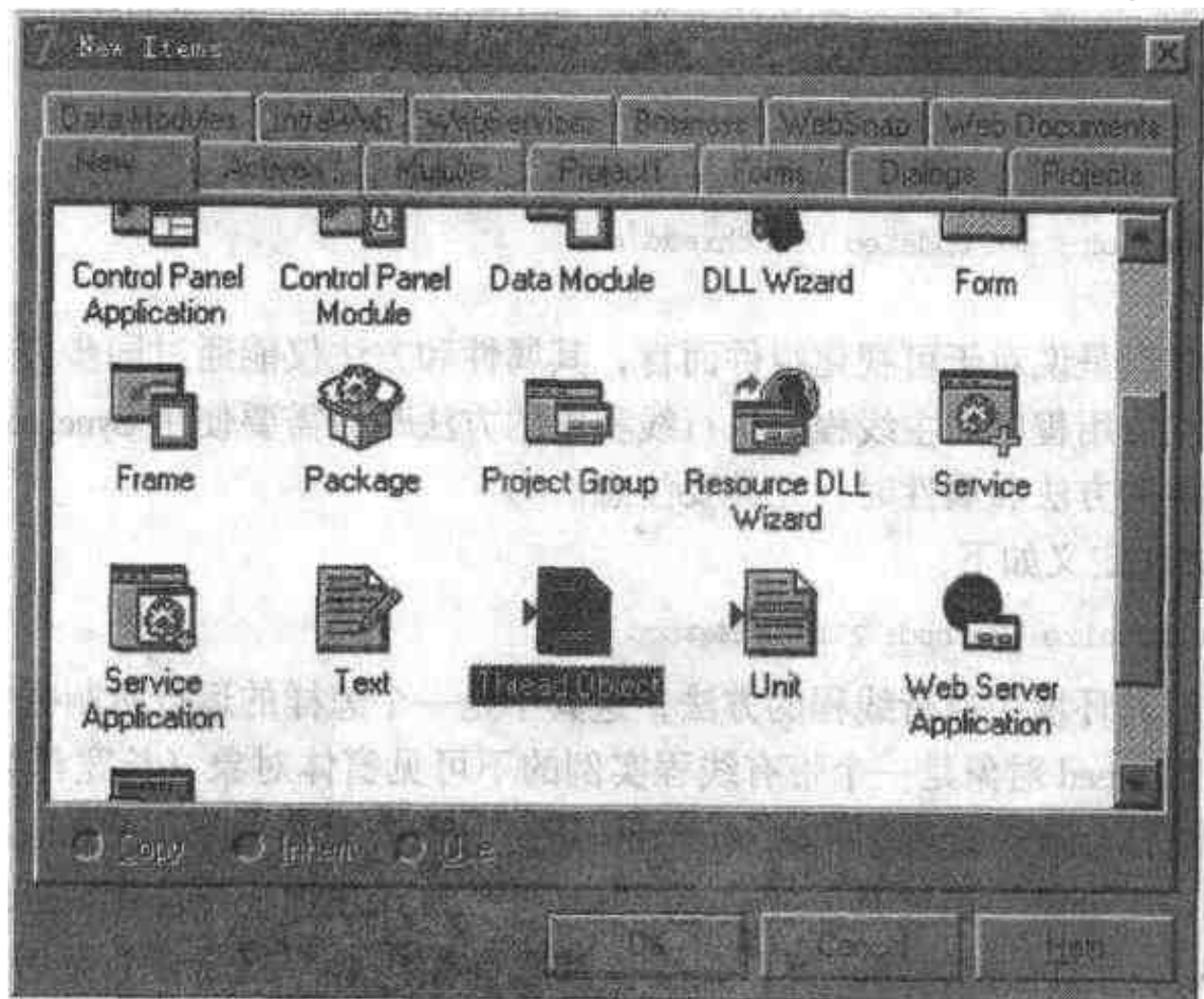


图 10-1 在 New Items 对话框中选择 Thread Object

选中了 New Items 对话框的 Thread Object 之后, 会弹出一个对话框要求我们输入新对象 (即 TThread 的派生类) 的名称。如果输入 TMyThread, Delphi 会自动创建一个包含 TMyThread 对象的新单元, 并有以下声明:

```

type
    TMyThread = class (TThread)
    private
        |Private declarations |
    protected
        procedure Execute; override;
    end;

```

从中可以看到, `Execute` 方法必须覆盖, 以实现 `TThread` 派生类的功能。而用于执行该线程的代码应该写在这个方法中。

```
procedure TMyThread.Execute;  
begin  
    {执行该线程的代码应该写在这里}  
end;
```

由此可见, 在 Delphi 中生成一个多线程应用程序最简单的办法是编写一个从 `TThread` 继承而来的线程类, 并通过覆盖 `Execute` 方法来完成线程的工作。当 `Execute` 结束时, 线程也就结束了。任何线程都可以简单地通过生成自定义线程类的一个实例来生成任何其他线程。每一个类的实例都作为一个独立的线程运行, 具有自己的堆栈。

大家可能会注意到, Delphi 自动创建的包含线程对象的新单元实现部分有以下一段注释:

```
{Important: Methods and properties of objects in visual components can only be used in a method  
called using Synchronize, for example, Synchronize (UpdateCaption); and UpdateCaption could look  
like,
```

```
procedure TMyThread.UpdateCaption;  
begin  
    Form1.Caption := 'Updated in a thread';  
end; }
```

这段注释的意思是说对于可视化组件而言, 其属性和方法仅能通过同步方法 `Synchronize` 来调用。这就是说在应用程序的主线程内执行线程中的方法时, 需要使用 `Synchronize` 方法, 特别是使用到可视组件的方法和属性时, 尤其要注意。

`Synchronize` 方法定义如下:

```
procedure Synchronize (Method: TThreadMethod);
```

它可以在主线程环境中执行线程的方法, 这其中是一个怎样的运行机制呢?

准确地说, `TThread` 对象是一个带有线程实例的不可见窗体对象 (长宽都为 0), 可以把这个窗体叫做线程窗体。这个线程窗体由该 `TThread` 类的所有对象共享。`TThread` 在构造的时候判断线程是否第一次创建, 如果是, 就创建线程窗体, 然后增加线程计数, 最后才建立线程实例。同理, `TThread` 对象在销毁的时候, 先减少线程计数, 然后判断计数是否为 0, 如果是就销毁线程窗体。

为什么要建立一个线程窗体呢? 答案就是 `TThread` 中的同步方法 `Synchronize` 的需要。线程对象存取其他 VCL 的属性和方法时, 与其他线程的同步机制是通过消息队列来实现的。`Synchronize ()` 把 `Method` 参数中指定的方法存储在一个名为 `FMethod` 的私有域中。当线程函数执行 `Synchronize ()` 时, 它就向线程窗体发送一条 `CM_EXECPROC` 消息, 将线程对象句柄 `Self` 作为该消息的 `IParam` 值进行发送。当线程窗体的窗体进程接受到该消息后, 它通过 `IParam` 指定的 `TThread` 对象调用 `FMethod` 中指定的方法。由于线程窗体是在主线程环境中创建的, 因此它的窗体过程也是由主线程执行的, 如图 10-2 所示。

因为线程窗体是进程的一个窗体 (虽然它不可见), 所以发向线程窗体的消息都会进入进程消息队列, 而消息队列的串行处理的特性保证不会出现访问冲突。这是一个简单而有效的解

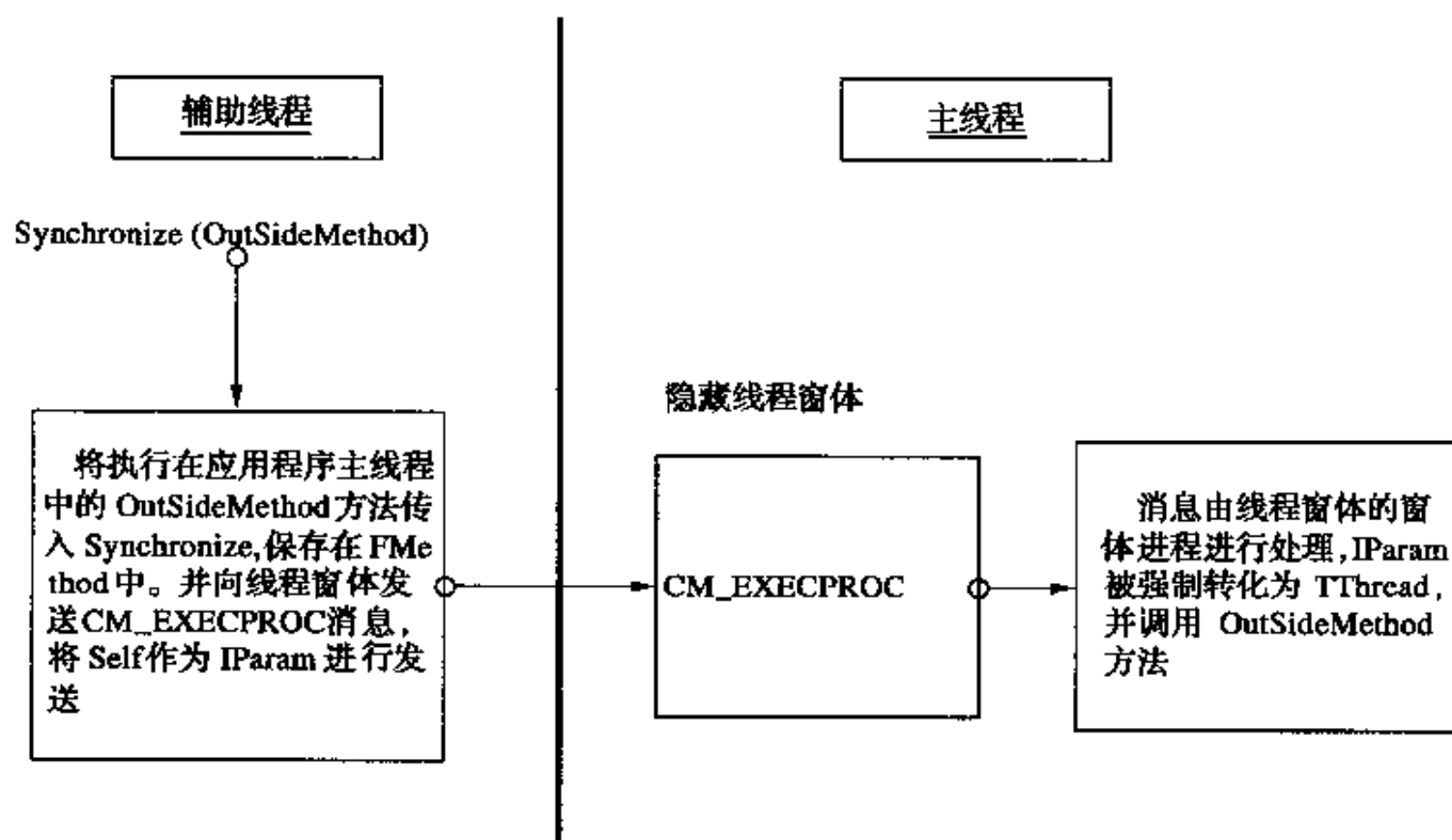


图 10-2 Synchronize 方法运行机制

决方案。如果在控制台程序中应用多线程, TThread 类可能就不太适合了。这种情况下要么直接应用线程函数, 要么自己写一个新的 TNewThread 类。

TThread 类在构造线程实例时没有直接调用 CreateThread () API 函数, 而是使用了一个 BeginThread () 函数。该函数的参数和 CreateThread () API 的一模一样的。这是一个让人振奋的地方, 因为 BeginThread () 加入了 Windows API 没有的异常处理功能。有意思的是, Delphi 在 BeginThread () 中创建了一个新的线程函数, 而把原来的线程函数和参数打包成 TThreadRec 作为新函数的参数。有关 Delphi 7 中 BeginThread 的定义如下:

```

type
  PThreadRec = ^TThreadRec;
  TThreadRec = record
    Func: TThreadFunc;
    Parameter: Pointer;
  end;

{$IFDEF MSWINDOWS}
function ThreadWrapper (Parameter: Pointer): Integer; stdcall;
{$ELSE}
function ThreadWrapper (Parameter: Pointer): Pointer; cdecl;
{$ENDIF}

asm
{$IFDEF PC_MAPPED_EXCEPTIONS}
  {Mark the top of the stack with a signature}
  PUSH    UNWINDFI_TOPOFSTACK
{$ENDIF}
  CALL    __FpuInit
  PUSH    EBP
{$IFDEF PC_MAPPED_EXCEPTIONS}
  XOR     ECX, ECX
  
```

```

        PUSH    offset _ExceptionHandler
        MOV     EDX, FS: [ECX]
        PUSH    EDX
        MOV     FS: [ECX], ESP
    { $ENDIF }
    { $IFDEF PC_MAPPED_EXCEPTIONS }
        // The signal handling code in SysUtils depends on being able to
        // discriminate between Delphi threads and foreign threads in order
        // to choose the disposition of certain signals. It does this by
        // testing a TLS index. However, we allocate TLS in a lazy fashion,
        // so this test can fail unless we've already allocated the TLS segment.
        // So we force the allocation of the TLS index value by touching a TLS
        // value here. So don't remove this silly call to AreOSExceptionsBlocked.
        CALL     AreOSExceptionsBlocked
    { $ENDIF }
        MOV     EAX, Parameter

        MOV     ECX, [EAX].TThreadRec.Parameter
        MOV     EDX, [EAX].TThreadRec.Func
        PUSH    ECX
        PUSH    EDX
        CALL     _FreeMem
        POP     EDX
        POP     EAX
        CALL     EDX

    { $IFDEF PC_MAPPED_EXCEPTIONS }
        XOR     EDX, EDX
        POP     ECX
        MOV     FS: [EDX], ECX
        POP     ECX
    { $ENDIF }
        POP     EBP
    { $IFDEF PC_MAPPED_EXCEPTIONS }
        { Ditch our TOS marker }
        ADD     ESP, 4
    { $ENDIF }
end;

{ $IFDEF MSWINDOWS }
function BeginThread (SecurityAttributes: Pointer; StackSize: LongWord;
    ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlags: LongWord;
    var ThreadId: LongWord): Integer;
var
    P: PThreadRec;
begin
    New (P);
    P.Func := ThreadFunc;
    P.Parameter := Parameter;
    IsMultiThread := TRUE;
    Result := CreateThread (SecurityAttributes, StackSize, @ThreadWrapper, P,
        CreationFlags, ThreadID);
end;

```

让人觉得美中不足的地方是 TThread 类在调用 BeginThread 时传递的 SercurityAttributes 和 StackSize 参数分别是 nil 和 0, 使 BeginThread () 在调用 CreateThread () 时使用了默认的安全设置和默认堆栈大小。有关这两个参数代表什么意义请查阅 Windows SDK 文档。

Delphi 调用 BeginThread 和 EndThread 来替代 Win32 API 的 CreateThread 和 ExitThread, 调用 BeginThread 的一个非常重要的作用就是将全局变量 IsMultiThread 设为 True, 因为 Delphi 的许多运行机制是当该变量为 True 时才是线程安全的, 例如 GetMem 和 FreeMem 函数。

10.1.3 使用线程对象

在 Delphi 中使用线程对象要比直接调用 Windows 的 API 函数来实现多线程更方便。线程对象作为一个封装好的类, 可以使我们在多线程编程中, 完全以面向对象的思维方式解决问题。本节我们重点讨论线程对象的使用以及线程同步问题。读者如果对 Delphi 中多线程编程的其他话题感兴趣, 则可以参阅我著的《Delphi 6 企业级解决方案及应用剖析》第 568 ~ 580 页 (机械工业出版社 2002 年出版)

在使用多线程对象开发应用程序中, 我们可以利用继承来实现不同的线程类。下面的示例程序 10-1 就利用了不同的线程类来完成多线程绘图。其中用于绘制圆形的 TCircleThread 和绘制方形 TRectangleThread 都是 TDrawThread 的派生类。在 TDrawThread 中, 我们定义了一个虚方法 Drawing (), 它是绘制图形的关键, 在派生类中它被覆盖, 用于绘制圆形或方形。绘图线程类 TDrawThread 继承自线程基类 TThread, 并覆盖了 Execute 方法。TThread 是抽象类, 无法直接实例化, 而 Execute () 方法是 TThread 的抽象方法, 必须覆盖。

示例程序 10-1 利用不同的线程类来完成多线程绘图

```
unit Thds;  
  
interface  
  
uses  
  Classes, Graphics, ExtCtrls;  
  
type  
  TDrawThread = class (TThread)  
    FCanvas: TCanvas;  
    FL, FT, FH, FW: Integer;  
  protected  
    procedure Drawing; virtual;  
    procedure Execute; override;  
  public  
    constructor Create (Box: TPaintBox);  
  end;  
  
  TCircleThread = class (TDrawThread)  
  public  
    procedure Drawing; override;  
  end;
```



```

TRectangleThread = class (TDrawThread)
public
    procedure Drawing; override;
end;

implementation

{TSortThread}

constructor TDrawThread.Create (Box: TPaintBox);
begin
    FL := Box.Left;
    FT := Box.Top;
    FH := Box.Height;
    FW := Box.Width;
    FCanvas := Box.Canvas;
    inherited Create (False);
end;

procedure TDrawThread.Execute;
begin
    FreeOnTerminate := True;
    synchronize (drawing);
end;

procedure TDrawThread.Drawing;
begin
    FCanvas.Brush.Style := bsClear;
end;

// 随机画圆
procedure TCircleThread.Drawing;
var
    x, y, z, i: integer;
begin
    inherited Drawing;
    for i := 0 to 10 do
    begin
        y := round (random (FH) + FT);
        x := round (random (FW) + FL);
        z := round (random (50));
        FCanvas.Pen.Color := clRed;
        FCanvas.Ellipse (x, y, (x+z), (y+z));
        // 模拟运算延时
        for y := 1 to 10000000 do z := round (random (sqr (x) * sqr (y))) );
    end;
end;

// 随机画方
procedure TRectangleThread.Drawing;
var
    x, y, z, i: integer;

```

```

begin
  inherited Drawing;
  for i := 0 to 10 do
  begin
    y := round (random (FH) + FT);
    x := round (random (FW) + FL);
    z := round (random (50));
    FCanvas.Pen.Color := clBlue;
    FCanvas.Rectangle (x, y, (x + z), (y + z));
    // 模拟运算延时
    for y := 1 to 10000000 do z := round (random (sqr (x * sqr (y))));
  end;
end;

end.

```

读者可能注意到 TDrawThread 的构造函数 Create (Box: TPaintBox) 中有以下一条语句:

```
inherited Create (False);
```

这是因为 TThread 的构造函数是这样声明的:

```
constructor Create (CreateSuspended: Boolean);
```

当 CreateSuspended 参数为 false 时, 表示线程对象的 Execute () 方法在执行 Create () 方法后立即自动执行, 否则需要在某个地方通过 Resume () 方法来启动这个线程。通常我们应该在 Create () 方法中设置好参数, 因为一旦线程运行起来再设置参数可能会引起错误。如果无法在 Create () 方法中设置好参数时, 可以先将 CreateSuspended 参数为 True, 待设置好参数后, 用 Resume () 方法激活线程。实际上, 运行一个线程, 需要调用 Delphi 运行时函数 BeginThread () 函数, 由它再调用 CreateThread () API 函数。其中 CreateSuspended 参数表示是否传入 CREATE_SUSPENDED 标志到 CreateThread () 函数中。

我们再看 TDrawThread 的 Execute () 方法实现代码。其中将线程对象的 FreeOnTerminate 属性设为 True, 意味着当程序使用完该线程对象后, 能自动销毁该对象, 释放内存。另外, synchronize (drawing) 这条语句使用了线程同步的机制。前面说过, VCL 的属性和方法只能在主线程中访问, 而在应用程序的主线程内执行线程中的方法时, 需要使用 Synchronize () 方法。

示例程序 10-2 是多线程绘图程序的界面单元, 我们在这个窗体上提供了绘图区, 以及主线程绘图和辅线程绘图的按钮, 所绘图形不同的形状和颜色能够帮助我们直观地了解不同线程的运行情况。最后运行界面应该如图 10-3 所示, 红色圆形和蓝色方形分别是辅线程 TCircleThread 和 TrectangleThread 的运行结果, 黑色椭圆应该是主线程运行结果。

示例程序 10-2 多线程绘图程序界面单元

```

unit frmThds;

interface

uses

```

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, ExtCtrls, Buttons;

type

```
TForm1 = class (TForm)
  PaintBox1: TPaintBox;
  btnThdDraw: TButton;
  btnClean: TButton;
  Bevel1: TBevel;
  btnMainDraw: TButton;
  btnExit: TButton;
  procedure btnExitClick (Sender: TObject);
  procedure btnMainDrawClick (Sender: TObject);
  procedure btnThdDrawClick (Sender: TObject);
  procedure btnCleanClick (Sender: TObject);
private
  {Private declarations }
public
  {Public declarations }
end;
```

var

```
Form1: TForm1;
```

implementation

uses Thds;

```
{ $R *. dfm }
```

```
procedure TForm1. btnExitClick (Sender: TObject);
begin
  close;
end;
```

//随机椭圆

```
procedure TForm1. btnMainDrawClick (Sender: TObject);
var
  x, y, z, i: integer;
begin
  for i := 0 to 10 do
  begin
    y := round (random (300) + PaintBox1. top);
    x := round (random (600) + PaintBox1. left);
    PaintBox1. Canvas. Pen. Color := clBlack;
    PaintBox1. Canvas. Ellipse (x, y, x + round (random (200)),
      y + round (random (200)));
    for y := 1 to 1000000 do z := round (random (sqr (x * sqr (y)))) ; //模拟运算延时
  end;
end;
```

```
procedure TForm1. btnThdDrawClick (Sender: TObject);
begin
  //创建线程
```

```
TRectangleThread. create (PaintBox1);  
TCircleThread. create (PaintBox1);  
end;  
  
procedure TForm1. btnCleanClick (Sender: TObject);  
begin  
    PaintBox1. Canvas. Brush. Style: = bsSolid;  
    PaintBox1. Canvas. Brush. Color: = PaintBox1. Color;  
    PaintBox1. Canvas. FillRect (PaintBox1. BoundsRect);  
end;  
  
end.
```

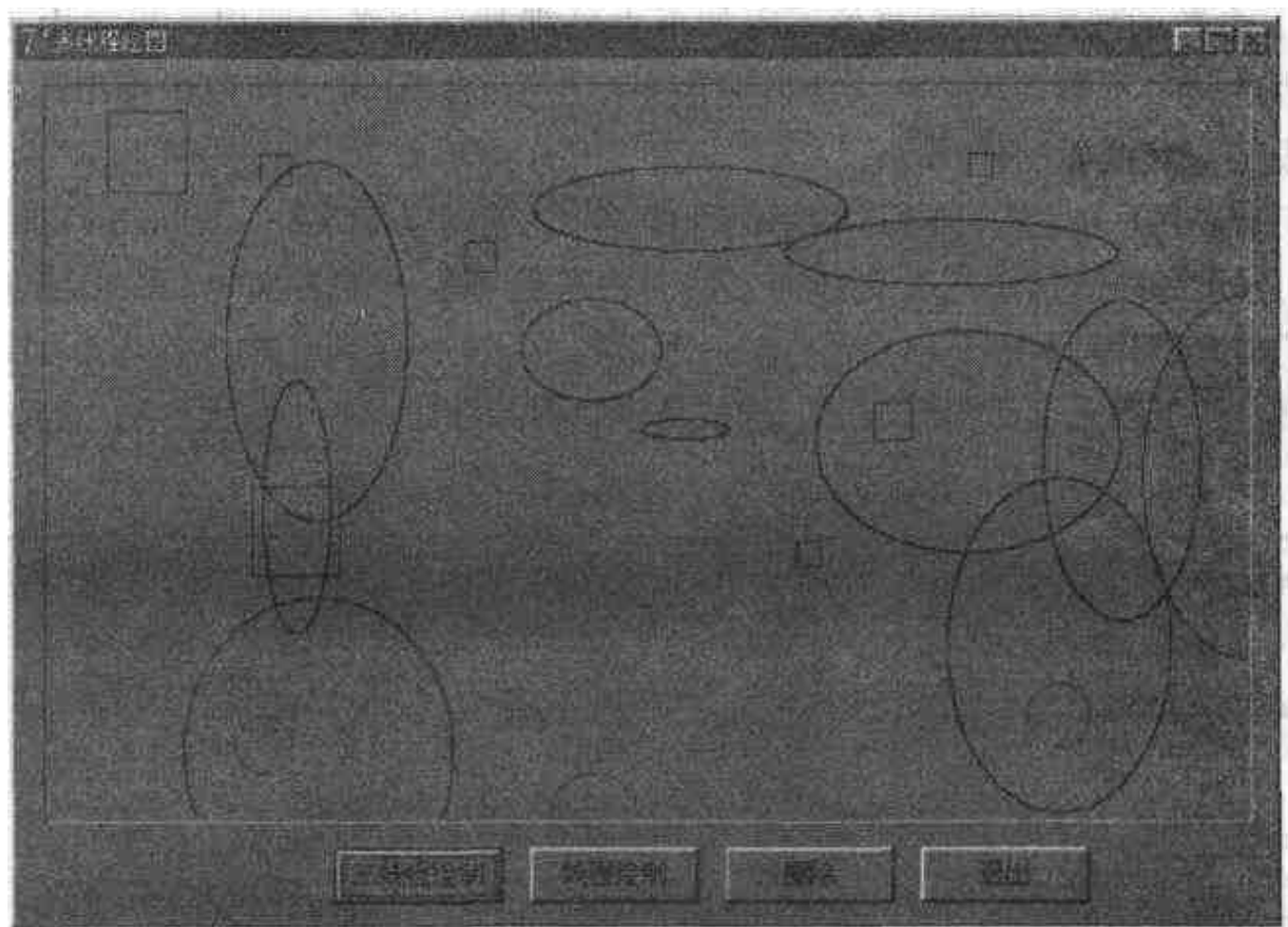


图 10-3 多线程绘图程序运行界面

为了使得观察准确，我们有意在绘图过程程序中加进了用于模拟运算延时的代码。因为如果绘图速度很快，我们将看不出绘图的各个线程是不是各自独立运行（读者不妨将该部分代码注释掉进行对比）。

我们点击线程绘制按钮，程序执行 `btnThdDrawClick` 事件，其代码创建线程对象后自动执行线程任务。可是运行的情况与我们预料的相反，我们没有看到绘图线程各自分别画出圆形和方形，而是先画方形，再画圆形。另外，运行绘图线程时，也无法点击其他按钮，执行主线程的任务。仔细查看程序，完全符合多线程编程的要求，难道多线程是假的？

虽然利用线程对象实现多线程编程在 Delphi 中已经不是很难，大多数介绍多线程编程的书也能找到一两个例子，但要真正弄懂并掌握却又是另一回事。上面的例子就是 Delphi 程序员在多线程编程中经常犯的错误，只不过测试时数据量小运算速度很快，不易发现而已。

这个问题的关键在于 `synchronize()` 方法的使用。前面我们讲过，为了保护一个 VCL 访问或

对 Windows GUI 的调用时, 可以用 `synchronize ()` 方法实现同步。该方法把过程作为参数, 并以线程安全的方式调用该过程。`synchronize ()` 方法把当前线程悬挂起来, 并让主线程调用这个过程, 如图 10-2 所示。当该过程结束时, 控制权返回到当前线程。因为所有对 `synchronize ()` 方法的调用都是由主线程处理的, 所以它们都受到避免竞争的保护。如果多个线程同时调用 `synchronize ()` 方法, 它们就必须排队等待, 因为同一个时间只有一个线程可获得对主线程的访问。也就是说并行的方法调用都转变为了串行的方法调用。

所以, 一旦创建了 `TRectangleThread` 线程对象, 示例程序 10-1 中 `synchronize (drawing)` 将先画完所有的方形, 才能释放同步资源, 将控制权返回到当前主线程, 然后接着调入 `TCircleThread` 线程对象画圆形。当然在主线程调用 `synchronize ()` 方法时, 它只能执行作为 `synchronize ()` 参数的那些方法, 当然不可能再去处理其他任务。所以, 程序中虽然实现了多线程, 但线程的同步不当造成一次同步就顺序完成了所有的任务, 使得线程串行执行而不是并行执行, 没发挥出多线程的作用。

为此我们需要将示例程序 10-1 改为示例程序 10-3, 使得 `synchronize (drawing)` 时只画一个图, 因为同步时间越短越好。这里无需更改原来的多线程绘图程序的界面单元。重新编译后运行程序, 可以看到画方形和画圆形是以并行方式一起进行的, 实现了真正的多线程。如果同时点击主线程绘制按钮, 还可以并行地画椭圆形。这就是多线程的好处, 各个线程分别运行, 互不影响。

示例程序 10-3 改进线程同步后的程序

```

interface
    uses
        Classes, Graphics, ExtCtrls;

    type
        TDrawThread = class (TThread)
            FCanvas: TCanvas;
            FL, FT, FH, FW: Integer;
            x, y, z: Integer;
        protected
            procedure Drawing; virtual;
            procedure Execute; override;
        public
            constructor Create (Box: TPaintBox);
        end;

        TCircleThread = class (TDrawThread)
        public
            procedure Drawing; override;
        end;

        TRectangleThread = class (TDrawThread)
        public
            procedure Drawing; override;
        end;

implementation

```

```

constructor TDrawThread.Create (Box: TPaintBox);
begin
    FL := Box.Left;
    FT := Box.Top;
    FH := Box.Height;
    FW := Box.Width;
    FCanvas := Box.Canvas;
    FreeOnTerminate := True;
    inherited Create (False);
end;

procedure TDrawThread.Execute;
var
    i, j: Integer;
begin
    for i := 1 to 10 do
        begin
            y := round (random (FH) + FT);
            x := round (random (FW) + FL);
            z := round (random (50));
            synchronize (drawing); //同步时间越短越好
            //模拟运算延时
            for j := 1 to 10000000 do z := round (random (sqr (x * sqr (j)))));
        end;
    end;

procedure TDrawThread.Drawing;
begin
    FCanvas.Brush.Style := bsClear;
end;

//随机画圆
procedure TCircleThread.Drawing;
begin
    inherited Drawing;
    FCanvas.Pen.Color := clRed;
    FCanvas.Ellipse (x, y, (x + z), (y + z));
end;

//随机画方
procedure TRectangleThread.Drawing;
begin
    inherited Drawing;
    FCanvas.Pen.Color := clBlue;
    FCanvas.Rectangle (x, y, (x + z), (y + z));
end;

end.

```

由于 VCL 并不是为多线程同时操作而设计的，因此需要引入强制同步机制 `synchronize` 来保证线程安全。不过，这里再次说明的是 VCL 中整个 Graphics 单元中的绘图对象（如：TCanvas、TPen、TBrush、TFont、TIcon 等）都是线程安全的，这就使得我们能够无需使用 `synchronize` 方法

也可以实现多线程同时操作单个图形对象。此时，为了保护线程对画布（Canvas）等图形对象的访问，也需要锁定一些相关的系统资源。幸运的是 TCanvas 提供了 Lock（）和 Unlock（）方法，使得我们能让线程中操作图形对象的那段代码处于保护之中，这就把使用 Synchronize 的做法改成了用事件（Event）和临界区（CriticalSection）的配合来进行同步，实现多线程对 VCL 图形组件的访问。因为 Lock（）方法代码类似于临界区和 CriticalSection（）API 函数的用法。

现在我们只要将示例程序 10-1 简单地改为示例程序 10-4，照样可以很好地实现多线程绘图。对于线程安全的某些 VCL 组件，不使用 synchronize 的做法给编程带来了很大的灵活性。记住，使用 synchronize 同步过程时，它是从主线程调用的，并不是真正的多线程执行。

示例程序 10-4 不使用 synchronize 的多线程绘图程序

```
unit Thds;

interface

uses
  Classes, Graphics, ExtCtrls;

type
  TDrawThread = class (TThread)
    FCanvas: TCanvas;
    FL, FT, FH, FW: Integer;
  protected
    procedure Drawing; virtual;
    procedure Execute; override;
  public
    constructor Create (Box: TPaintBox);
  end;

  TCircleThread = class (TDrawThread)
  public
    procedure Drawing; override;
  end;

  TRectangleThread = class (TDrawThread)
  public
    procedure Drawing; override;
  end;

implementation

constructor TDrawThread.Create (Box: TPaintBox);
begin
  FL := Box.Left;
  FT := Box.Top;
  FH := Box.Height;
  FW := Box.Width;
  FCanvas := Box.Canvas;
  inherited Create (False);
end;
```



```

procedure TDrawThread.Execute;
begin
    FreeOnTerminate := True;
    //synchronize (drawing); 没有使用 VCL 的同步机制
    drawing;
end;

procedure TDrawThread.Drawing;
begin
    FCanvas.Brush.Style := bsClear;
end;

//随机画圆
procedure TCircleThread.Drawing;
var
    x, y, z, i: integer;
begin
    inherited Drawing;
    for i := 0 to 10 do
    begin
        y := round (random (FH) + FT);
        x := round (random (FW) + FL);
        z := round (random (50));
        FCanvas.lock; //加锁
        FCanvas.Pen.Color := clRed;
        FCanvas.Ellipse (x, y, (x + z), (y + z));
        FCanvas.unlock; //解锁
        //模拟运算延时
        for y := 1 to 10000000 do z := round (random (sqr (x * sqr (y)))));
    end;
end;

//随机画方
procedure TRectangleThread.Drawing;
var
    x, y, z, i: integer;
begin
    inherited Drawing;
    for i := 0 to 10 do
    begin
        y := round (random (FH) + FT);
        x := round (random (FW) + FL);
        z := round (random (50));
        FCanvas.lock; //加锁
        FCanvas.Pen.Color := clBlue;
        FCanvas.Rectangle (x, y, (x + z), (y + z));
        FCanvas.unlock; //解锁
        //模拟运算延时
        for y := 1 to 10000000 do z := round (random (sqr (x * sqr (y)))));
    end;
end;

end.

```

10.2 TStrings、TList、TCollection: 列表与集合

在编程中常常需要管理大量的字符串、对象等,除了可用动态数组管理这些对象和字符串外,还可用 TStrings、TList 或 TCollection 进行管理。动态数组与 TStrings、TList 和 TCollection 的主要区别在于,动态数组只适用于管理单一数据类型,且数组的维数扩展要由程序来完成,但在程序编译时,编译器将对程序进行严格的类型检查。TStrings 的派生类主要用于管理字符串集合和对象集合,如 TStringList; TList 类主要用于管理对象集,因为它内部管理的是一个对象引用数组;而 TCollection 类主要用于管理由 TCollectionItem 派生类的集合。从三个类的适用范围看, TStrings 最广,而 TList 次之。使用 TStrings 的派生类既可管理字符串集,又可管理对象集。

10.2.1 TStrings 与 TStringList

TStrings 是一个抽象类,抽象类在类中定义有抽象方法,这些抽象方法要由派生类声明相应的覆盖函数来实现,因此不能创建抽象类的实例。TStringList 类是 TStrings 类的派生类,常作为组件的属性。TStrings 继承于 TPersistent 类,该类之所以从 TPersistent 继承而不从 TObject 类继承的最主要原因是为了能在 Object Inspector 中设置对象的有关属性。在一些 VCL 组件中都设有属于 TStrings 类的属性,如 TListBox.Items、TQuery.Params 等,可在属性编辑窗口中轻松设置这些属性的字符串。由于不能创建 TStrings 抽象类的实例,因此要用如下方法来将类实例化。代码类似如下:

```
Type
  TMyObject = Class (TComponent)
  Private
    FItems: TStrings;
    {.....}
  Published
    Constructor Create (AOwner: TComponent)
  Property Items: TStrings read FItems write FItems;
End;

Implementation

Constructor TMyObject.Create (AOwner: TComponent)
Begin
  Inherited Create (AOwner);
  FItems := TStringList.Create;
  {.....}
end;
```

从代码中可以看出,组件的属性往往声明为 TStrings 类型,而在组件初始化时,赋值给该属性的是 TStrings 的派生类所创建的对象。这样做的原因是 TStrings 类已为其派生类声明了各种操作所需的方法和属性,派生类只要根据具体的需要实现所需的方法和属性,在派生类中覆盖抽象方法即可。许多 VCL 组件都有 TStrings 类的属性,属性以 TStrings 类声明有利于利用多态来保持属性统一,使用户无需了解各派生类的结构却可使用对象;派生类的相关代码的编写交由组件编写者来完成。

TStringList 类是 TStrings 的派生类, 它在内部分别为字符串和对象各声明了一个数组来存储数据, 因此 TStringList 能同时保存字符串和对象。使用 TStrings 来管理字符串的最大好处就是类字符串或类的操作提供了加入 (Add)、插入 (Insert)、删除 (Delete) 和查找索引 (indexOf) 等方法, 同时可在 Object Inspector 中设置字符串的初始值, 使应用程序的代码更简单。另外它还覆盖了基类 TStrings 的两个重要的方法 LoadFromFile 和 SaveToFile, SaveToFile 可将 TStrings 所存的字符串存成文件, 而 LoadFromFile 可将文件中的字符串读入。同样也可将集合中的值写到流或从流中读出。示例程序 10-5 以对象集的管理来说明 TStringList 类的使用 (见图 10-4)。

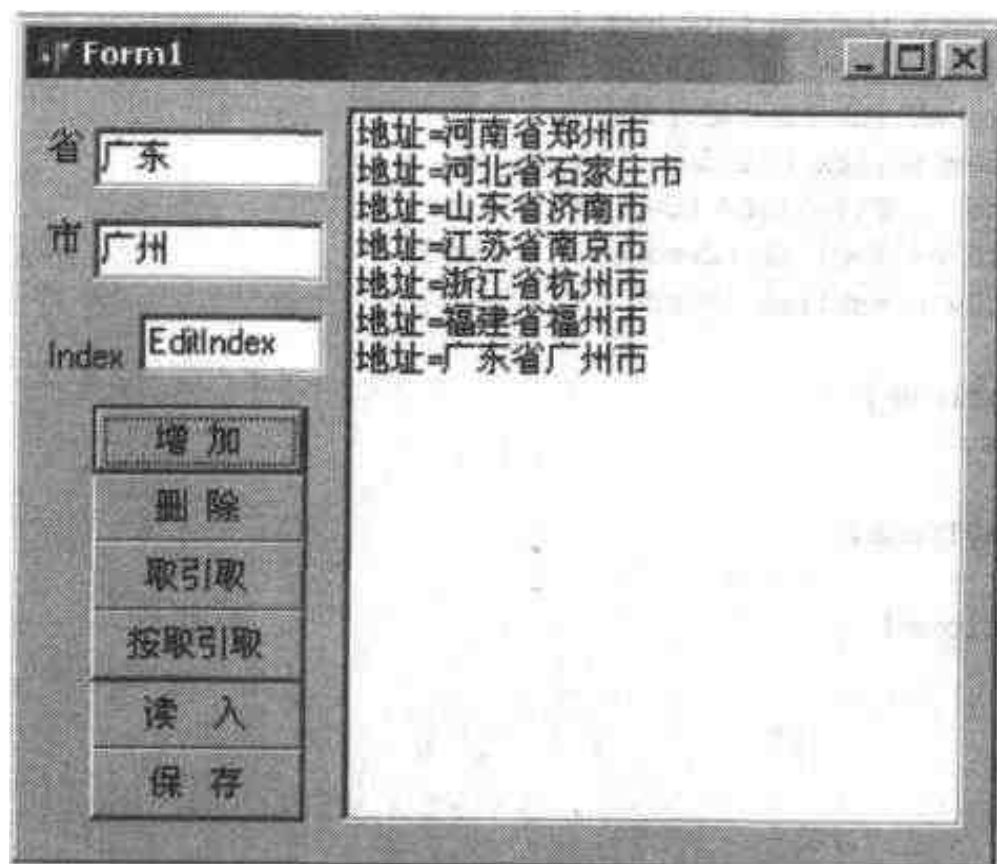


图 10-4 TStringList 对象的使用范例

示例程序 10-5 TStringList 对象的使用范例源程序

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TAddress = Class (TObject)
  Private
    FProvince: String;
    FCity: String;
  public
    Constructor Create (pProvince, pCity: String);
    Property Province: String read FProvince write FProvince;
    Property City: String Read FCity Write FCity;
  end;

  TForm1 = class (TForm)
    BtnAdd: TButton;
    BtnDelete: TButton;
    BtnIndexOf: TButton;
```

```

    BtnLoadFromFile: TButton;
    BtnSaveToFile: TButton;
    ListBox1: TListBox;
    EditCity: TEdit;
    EditProvince: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    EditIndex: TEdit;
    BtnGetByIndex: TButton;
    Label3: TLabel;
    procedure FormCreate (Sender: TObject);
    procedure BtnAddClick (Sender: TObject);
    procedure BtnDeleteClick (Sender: TObject);
    procedure BtnIndexOfClick (Sender: TObject);
    procedure BtnLoadFromFileClick (Sender: TObject);
    procedure BtnSaveToFileClick (Sender: TObject);
    procedure BtnGetByIndexClick (Sender: TObject);
  private
    {Private declarations}
    FItems: TStrings;
    Obj: TObject;
    procedure DisplayItems;
  public
    {Public declarations}
  end;

var
    Form1: TForm1;

implementation

{$R *.dfm}

{TAddress}
constructor TAddress.Create (pProvince, pCity: String);
begin
    Inherited Create;
    FProvince: = pProvince;
    FCity: = pCity;
end;

procedure TForm1.FormCreate (Sender: TObject);
begin
    FItems: = TStringList.Create;
end;

procedure TForm1.BtnAddClick (Sender: TObject);
begin
    Obj: = TAddress.Create (EditProvince.Text, EditCity.Text);
    FItems.AddObject (EditCity.Text, Obj);
    DisplayItems;
end;

procedure TForm1.BtnDeleteClick (Sender: TObject);

```

```
begin
  FItems.Delete (StrToInt (EditIndex.text) );
  DisplayItems;
end;

procedure TForm1.BtnIndexOfClick (Sender: TObject);
var
  Index: integer;
begin
  Index: = FItems.IndexOfObject (Obj);
  Caption: = Format (' Index = %d', [Index] );
end;

procedure TForm1.BtnLoadFromFileClick (Sender: TObject);
begin
  FItems.Clear;
  FItems.LoadFromFile (' StringsData.dat ');
  DisplayItems;
end;

procedure TForm1.BtnSaveToFileClick (Sender: TObject);
begin
  FItems.SaveToFile (' StringsData.dat ');
end;

procedure TForm1.DisplayItems;
Var
  i: integer;
  Str: String;
  Address: TAddress;
begin
  ListBox1.Clear;
  For i: = 0 to FItems.Count-1 do
  begin
    Address: = FItems.Objects [i] as TAddress;
    Str: = Format ('地址 = %s省%s市', [Address.Province, Address.City] );
    ListBox1.Items.Add (Str);
  end;
end;

procedure TForm1.BtnGetByIndexClick (Sender: TObject);
Var
  Address: TAddress;
begin
  Address: = FItems.Objects [StrToInt (EditIndex.text)] as TAddress;
  Caption: = Format ('地址 = %s省%s市', [Address.Province, Address.City] );
end;

end.
```

从示例程序 10-5 的代码中可见, 在用 AddObject 向 TStringList 加入对象时, 同时要加入一个与对象相关联的字符串, 即对象的名, 用此字符串可查找对象。在用 IndexOfObject 方法查对象的索引时, 该方法是用对象的指针来判断的, 而不是用对象的属性值来查找, 因此在实际应

用中可用对象名查找索引，然后由索引定位对象。在用 TStringList 管理对象时，LoadFromFile 和 SaveToFile 这两个方法比较特殊，即在用 SaveToFile 将 TStringList 中的对象保存到文件时，实际上只保存了对象的名，而对象的属性并未存入文件；同样用 LoadFromFile 读入时，只读入对象的名，对象并没被创建。因此，例子程序中按下“读入”按钮时将会出错。例子之所以保留该错误就是为引起大家的注意。

由于 TStringList 类同时可保存字符串和对象，而且所管理的对象有名字，可通过对象名来操作对象，因此这种结构特别适合做哈希表，有兴趣的读者可自己试用 TStringList 写一个哈希表程序。

10.2.2 TList

对象集的管理，若不是哈希表，用 TList 可能更方便。TList 提供了增加、删除等操作，但与 TStringList 有些不同。我们仍以前面 TStringList 例子所完成的功能来说明。当要向 TList 增加多条记录时，可用如下的代码来提高程序的运行效率。

```
Var
  i: Integer;
  Obj: TAddress;
begin
  List.Capacity: = 10;
  For i: = 0 to 9 do
  begin
    Obj: = TAddress.Create ('湖北省武汉', format ('XX街%d号', [i]));
    List.Add (Obj);
  end;
end;
```

在增加多条记录时，上面程序的运行效率比省却 List.Capacity: = 10 的效率高。Capacity 的值大于等于 Count，当 Capacity 与 Count 相等而给 TList 增加一条记录时，TList 对象需要加长内部保存对象的数组。在设 Capacity 的值时，对象将一次加大数组的长度，这样在增加对象时，由于 Count 小于 Capacity 因此不必每次都需要加长数组而分配内存。Expand 方法的作用与 Capacity 相似，它是将数据的长度在现有长度上加长指定数量。

TList 的 Exchange 方法可用于交换两指定位置的对象在数组中的位置，如：

```
List.Exchange (0, 10);
```

将把 List 中第 1 个对象和第 11 个对象的位置相交换。Extract 与 Delete 两个方法都是从 TList 中删除一个对象，但却有很大的不同，Delete 方法是以对象在 TList 中的索引号来指定要删除的对象；而 Extract 是用对象来指定要删除的元素。在 Extract 的内部，它是通过比较对象的指针来定位要删除的对象。一方面为了节约存储空间，另一方面也为了编程方便，TList 还提供了一个删除所存储的指针为 nil 的项。

TList 所存储的是对象的指针，并且函数的参数也声明为 Pointer 数据类型，因此 TList 也可用于管理记录和字符串，不过在操作上要转换为指针，如：

```
type
  TAddress = record
```

```

        Province: String;
        City: String;
    End;

Var
    Address: TAddress;
Begin
    Address.Province: = '湖北省';
    Address.City: = '武汉市';
    List.Add (Addr (Address) );
End;

Var
    Str: String;
Begin
    Str: = '湖北省武汉市';
    List.Add (PChar (Str) );
End;

```

TList 存储和管理对象的强大功能在其派生类 TObjectList 以及 TObjectList 的派生类 TComponentList 中得到了很好的体现。所以在使用对象集时, 我们应该优先考虑使用这些 VCL 对象。

参见 关于 TObjectList 的介绍请参见 4.3 节。

10.2.3 TCollection

TCollection 用于管理 TCollectionItem 或其派生类的对象集, 也就是说它是 TCollectionItem 或其派生类的容器。它们都是 TPersistent 类的派生类, 所以类型的属性可在 Object Inspector 中显示。假如要管理一个学生档案的集合, 首先就要从 TCollectionItem 派生一个用于保存学生档案的派生类, 定义如下:

```

TStudent = Class (TCollectionItem)
Private
    FID: string;
    FName: String;
    FTel: String;
    FRoomNum: String;
Public
    Property ID: string read FID write FID;;
    Property Name: String read FName write;
    Property Tel: String read FTel write FTel;
    Property RoomNum: String read FRoomNum Write FRoomNum;
end;

```

在窗口创建时要创建对象, 创建的过程为:

```

procedure TForm1.FormCreate (Sender: TObject);
begin
    FCollection: = TCollection.Create (TStudent);
end;

```


要向集合加入新对象而调用 Add 方法时, TCollection 对象将创建一个新对象, 并返回新对象的引用, 程序可通过该对象引用设置对象的属性。其他方法的调用都与此类似。具体见示例程序 10-6。运行结果如图 10-5 所示。

图 10-5 TCollection 对象使用范例运行结果

示例程序 10-6 TCollection 对象使用范例的源程序

```
unit Main;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons;
type
  TStudent = Class (TCollectionItem)
  Private
    FID: string;
    FName: String;
    FTel: String;
    FRoomNum: String;
  Public
    Property ID: string read FID write FID;
    Property Name: String read FName write FName;
    Property Tel: String read FTel write FTel;
    Property RoomNum: String read FRoomNum Write FRoomNum;
  end;

  TForm1 = class (TForm)
    EditID: TEdit;
    EditName: TEdit;
    EditTel: TEdit;
    EditRoomNum: TEdit;
    BtnAdd: TButton;
    BtnDelete: TButton;
    StaticText1: TStaticText;
    StaticText2: TStaticText;
```

```

    StaticText3: TStaticText;
    StaticText4: TStaticText;
    BtnDisplay: TBitBtn;
    procedure FormCreate (Sender: TObject);
    procedure BtnAddClick (Sender: TObject);
    procedure BtnDeleteClick (Sender: TObject);
    procedure BtnDisplayClick (Sender: TObject);
private
    {Private declarations}
    FCollection: TCollection;
public
    {Public declarations}
end;

var
    Form1: TForm1;

implementation
{$R *.dfm}

{TForm1}
procedure TForm1.FormCreate (Sender: TObject);
begin
    FCollection := TCollection.Create (TStudent);
end;

procedure TForm1.BtnAddClick (Sender: TObject);
Var
    Student: TStudent;
begin
    Student := (FCollection.Add) as TStudent;
    Student.FID := EditID.Text;
    Student.FName := EditName.Text;
    Student.Tel := EditTel.Text;
    Student.RoomNum := EditRoomNum.Text;
end;

procedure TForm1.BtnDeleteClick (Sender: TObject);
begin
    FCollection.Delete (0);
end;

procedure TForm1.BtnDisplayClick (Sender: TObject);
Var
    Student: TStudent;
begin
    Student := FCollection.Items [0] as TStudent;
    EditID.Text := Student.FID;
    EditName.Text := Student.FName;
    EditTel.Text := Student.Tel;
    EditRoomNum.Text := Student.FRoomNum;
end;

end.

```

如果 TCollection 是作为组件的 published 属性时, 要使该属性能在 Object Inspector 中进行运行期设计, 则在对象创建后还需设置对象的 PropName 属性, 所设的值将作为该对象所表示的属性名出现在 Object Inspector 左边列表栏中。

为了能够操作记录、数据类型、对象甚至其他组件, 组件通常需要建立一些列表。在某些情况下, 组件最好把列表封装成一个对象, 并且使这个对象成为组件的一个属性。例如, TMemo 组件的 Lines 属性是一个 TStrings 对象, 它封装了一个字符串列表。这样, 当用户存储数据时 TStrings 对象就负责为用户流化保存各行数据到窗体文件中。

如果列表没有被一个已存在的类 (如 TString) 封装成一个对象, 但又想存储这个列表, 这时该怎么办? 我们可以创建一个类, 使这个类成为组件的一个属性, 然后用这个类来控制列表。也可以覆盖默认的流程机制使它具有流化列表的作用。然而, 最好的解决方法是利用 TCollection 和 TCollectionItem 类。

TCollection 类是一个对象, 用于存储由 TCollectionItem 对象组成的集合, 它也是从 TPersistent 继承下来的一个组件。通常把 TCollection 和一个已存在组件关联起来。

若用 TCollection 来存储一个项目集合, 那么首先要从 TCollection 派生出一个类, 假定叫做 TNewCollection, 它可以作为组件的某种属性。然后从 TCollectionItem 派生出一个类, 假定叫做 TNewCollectionItem。TNewCollection 负责维护 TNewCollectionItem 对象的集合。这样做的好处是, 如果 TNewCollectionItem 的数据需要做流化处理, 则只要成为由 TNewCollectionItem 公开的属性即可。Delphi 已经知道如何流化公开属性。

使用 TCollection 的一个例子是我们常用的 TDBGrid 的列集合。TDBGrid 的 Columns 属性的类型是 TDBGridColumnColumns。TDBGridColumnColumns 就是从 TCollection 继承下来的, 具体声明如下:

```
TDBGridColumnColumns = class (TCollection)
private
    FGrid: TCustomDBGrid;
    function GetColumn (Index: Integer): TColumn;
    function InternalAdd: TColumn;
    procedure SetColumn (Index: Integer; Value: TColumn);
    procedure SetState (NewState: TDBGridColumnColumnsState);
    function GetState: TDBGridColumnColumnsState;
protected
    function GetOwner: TPersistent; override;
    procedure Update (Item: TCollectionItem); override;
public
    constructor Create (Grid: TCustomDBGrid; ColumnClass: TColumnClass);
    function Add: TColumn;
    procedure LoadFromFile (const Filename: string);
    procedure LoadFromStream (S: TStream);
    procedure RestoreDefaults;
    procedure RebuildColumns;
    procedure SaveToFile (const Filename: string);
    procedure SaveToStream (S: TStream);
    property State: TDBGridColumnColumnsState read GetState write SetState;
    property Grid: TCustomDBGrid read FGrid;
    property Items [Index: Integer]: TColumn read GetColumn
        write SetColumn; default;
```

```
end;
```

TDBGridColumns 存储了由 TCollectionItem 派生的列 TColumn 声明, 如下所示:

```
TColumn = class (TCollectionItem)
private
    FField: TField;
    FFieldName: string;
    FColor: TColor;
    .....
protected
    function CreateTitle: TColumnTitle; virtual;
    function GetGrid: TCustomDBGrid;
    .....
public
    constructor Create (Collection: TCollection); override;
    destructor Destroy; override;
    procedure Assign (Source: TPersistent); override;
    .....
published
    property Alignment: TAlignment read GetAlignment
        write SetAlignment stored IsAlignmentStored;
    property ButtonStyle: TColumnButtonStyle read FButtonStyle
        write SetButtonStyle default cbsAuto;
    property Color: TColor read GetColor write SetColor stored IsColorStored;
    property DropDownRows: Cardinal read FDropDownRows
        write FDropDownRows default 7;
    property Expanded: Boolean read GetExpanded
        write SetExpanded default True;
    property FieldName: String read FFieldName write SetFieldName;
    property Font: TFont read GetFont write SetFont stored IsFontStored;
    property ImeMode: TImeMode read GetImeMode
        write SetImeMode stored IsImeModeStored;
    property ImeName: TImeName read GetImeName
        write SetImeName stored IsImeNameStored;
    property PickList: TStrings read GetPickList write SetPickList;
    property PopupMenu: TPopupMenu read FPopupMenu write SetPopupMenu;
    property ReadOnly: Boolean read GetReadOnly
        write SetReadOnly stored IsReadOnlyStored;
    property Title: TColumnTitle read FTitle write SetTitle;
    property Width: Integer read GetWidth
        write SetWidth stored IsWidthStored;
    property Visible: Boolean read GetVisible write SetVisible;
end;
```

凡是在类的 published 部分声明的 TColumn 属性都由 Delphi 自动进行流操作。

TColumn 的 Create () 带一个 TCollection 类型的参数, 使自己依附于这个 TCollection 对象。同样, TStatusPanels 的 Create () 也带了一个 TStatusBar 组件作为参数, 使自己依附于这个对象。TCollection 知道该怎样对 TCollectionItem 对象进行流操作, 并且定义了一些属性和方法来操纵 TCollectionItem 对象。关于这些属性和方法的详细内容, 请查阅联机帮助。

需要说明的是 TCollection 和 TCollectionItem 的派生类在 VCL 组件中应用很广, 如表 10-1 所示。掌握它可以使我们在使用这些组件时得心应手, 提高效率。

表 10-1 涉及 TCollection 和 TCollectionItem 派生类的常用 VCL 组件

组 件	TCollection 的派生类	TCollectionItem 的派生类
TClientDataSet	TAggregates	TAggregate
TWebResponse	TCookieCollection	TCookie
TCoolBar	TCoolBands	TCoolBand
TDBGrid	TDBGridColumns	TColumn
TService	TDependencies	TDependency
TDecisionGrid	TDisplayDims	TDisplayDim
TDataSet	TFieldDefs	TFieldDef
THeaderControl	THeaderSections	THeaderSection
TTable	TIndexDefs	TIndexDef
TListView	TListColumns	TListColumn
许多数据集组件	TParams	TParam
TStatusBar	TStatusPanels	TStatusPanel
TListView	TWorkAreas	TWorkArea

10.3 TStream: 流对象与流化存储技术

流对象与流化存储技术不仅是 Delphi 可视化设计实现的核心, 它也为面向对象编程的数据存储提供了更为方便、简单和通用的方法。尽管流化存储所涉及的存储媒介十分广泛, 但在各对象的接口上得到了统一, 使程序的存储操作变得十分方便、简单, 从而使程序员能站在更高层面上进行数据存取的有关编程工作而无需考虑存储介质的具体差异。因此, 认识、理解和灵活运用 Delphi 的流类是 Delphi 程序员必须掌握的基础内容。本节首先介绍流的基类 TStream 及其派生类间的分类与继承关系, 然后介绍了用于文件操作的文件流 TFileStream、内存流 TMemStream 和数据压缩流 TCompressionStream 及 TDecompressionStream。最后介绍了与组件属性存储相关的 TReader 类和 TWriter 类, 以及流与 Delphi 可视化设计中的组件属性存取及恢复相关的底层知识。

10.3.1 TStream 类及其派生类

TStream 是一个抽象基类, 是所有 Stream 类的基类, 它继承自 TObject。TStream 类是抽象类, 随着存储媒介的不同, 它又派生出一些子类, 这些子类主要用于对文本、内存、数据库的 Blob 字段、数据压缩等进行操作。由于 TStream 与具体的存储无关, 派生类却与存储媒介紧密相关, 因此每个子类都必须实现与具体存储媒介相关的方法, 如磁盘、内存等。TStream 的派生类不仅能将数据以流的形式存储到具体的媒介中, 还能将 VCL 组件的属性存储到流中, 如 SpeedButton 的 Glyph, 它就是将按钮的图标以流的方式存储到窗口的 DFM 文件中。因此流的使用在 Delphi 中无所不在。

TStream 类的主要派生类及继承关系如图 10-6 所示。其中, TFileStream 是磁盘数据流化处理的派生类, 它将磁盘文件的存取以流的方式进行操作; TBlobStream 主要是对表中的 Blob 字段按流的方式进行存取操作; TOleStream 用于在 Com 中进行参数的传递; TMemoryStream 是以内存为存储媒介进行数据的流化存取; 而 TWinSocketStream 是一基于 Socket 接口的流化操作对象, 在编写以 Socket 进行数据传输的程序中特别有用。另外, 还有一对 TCompressionStream 和

TDecompressionStream 可以为基于 ZIP 压缩的数据存取提供流化操作手段。不过它不在 VCL 中, 而是在 CLX 中。

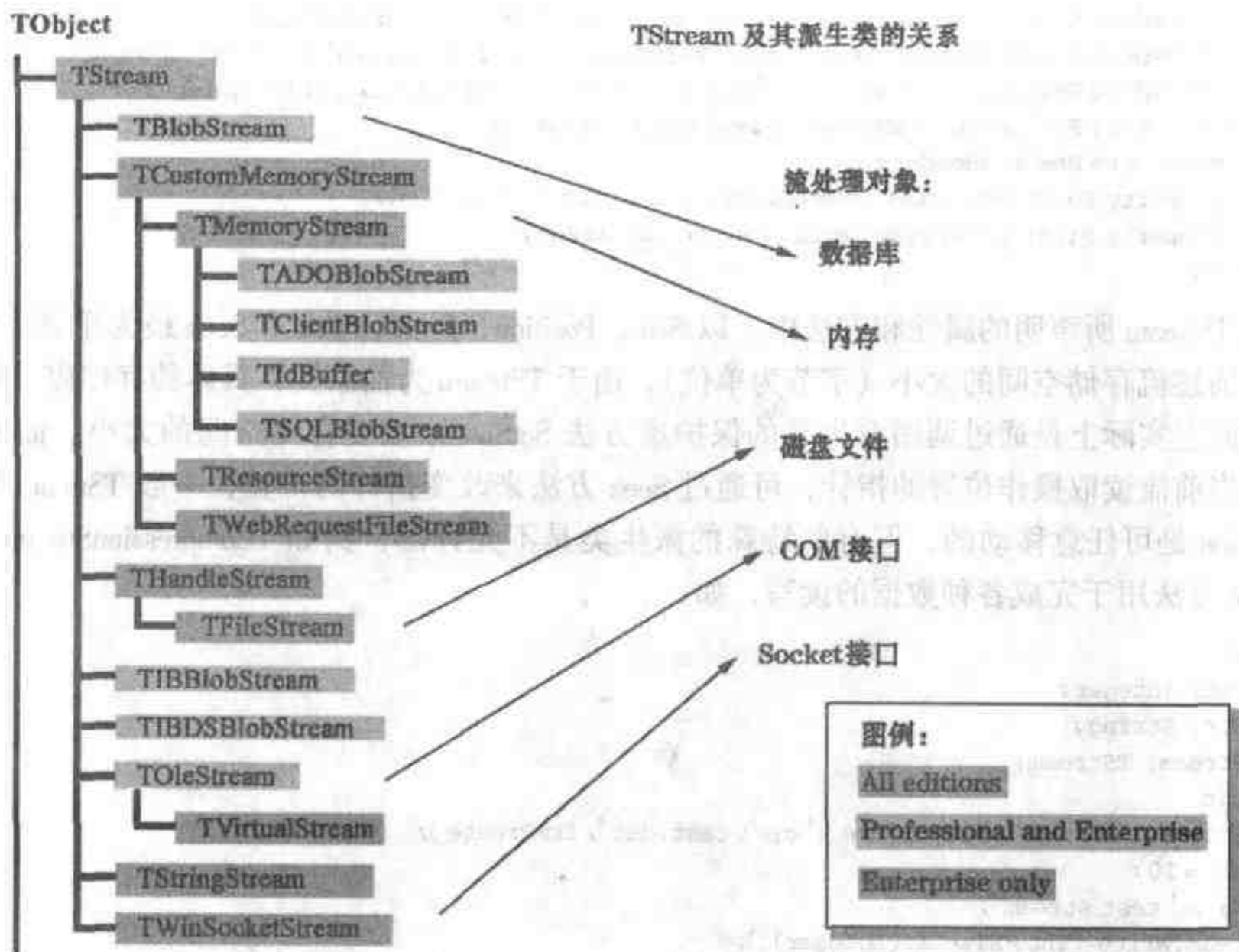


图 10-6 TStream 及其派生类的关系

我们知道, TStream 是所有 Stream 类的基类, 它也是一个抽象类, 因此不能直接用 TStream 的 Create 创建类的实例, 而只能创建其派生类的实例。TStream 类所声明的方法相当少, 而且简单, 主要是声明了一些 Stream 类所共有的抽象方法和虚方法, 这些方法需要在派生类中实现或覆盖。TStream 为数据在多种存储媒介中以二进制方式存取提供了统一的接口和底层支持。TStream 公有属性和方法声明如下:

```

TStream = class (TObject)
protected
  procedure SetSize (NewSize: Longint); overload; virtual;
  procedure SetSize (const NewSize: Int64); overload; virtual;
public
  function Read (var Buffer; Count: Longint): Longint; virtual; abstract;
  function Write (const Buffer; Count: Longint): Longint; virtual; abstract;
  function Seek (Offset: Longint; Origin: Word): Longint; overload; virtual;
  function Seek (const Offset: Int64; Origin: TSeekOrigin): Int64; overload; virtual;
  procedure ReadBuffer (var Buffer; Count: Longint);
  procedure WriteBuffer (const Buffer; Count: Longint);
  function CopyFrom (Source: TStream; Count: Int64): Int64;
  function ReadComponent (Instance: TComponent): TComponent;

```

```

function ReadComponentRes (Instance: TComponent): TComponent;
procedure WriteComponent (Instance: TComponent);
procedure WriteComponentRes (const ResName: string; Instance: TComponent);
procedure WriteDescendent (Instance, Ancestor: TComponent); virtual;
procedure WriteDescendentRes (const ResName: string; Instance, Ancestor: TComponent);
procedure WriteResourceHeader (const ResName: string; out FixupInfo: Integer);
procedure FixupResourceHeader (FixupInfo: Integer);
procedure ReadResHeader;
property Position: Int64 read GetPosition write SetPosition;
property Size: Int64 read GetSize write SetSize64;
end;

```

在 TStream 所声明的属性和方法中, 以 Size、Position、Read、Write、Seek 最为重要。Size 属性用于描述流存储空间的大小 (字节为单位), 由于 TStream 为抽象类, 具体的存储媒介还不确定, 因此它实际上是通过调用派生类的保护虚方法 SetSize 来改变存储空间的大小。而 Position 属性是当前流读取操作位置的指针, 可通过 Seek 方法来改变指针的位置, 一般 TStream 派生类的 Position 是可任意移动的, 但有些特殊的派生类是不允许的, 例如 TCompressionStream。Read 和 Write 方法用于完成各种数据的读写, 如:

```

var
  Int: integer;
  Str: String;
  Stream: TStream;
Begin
  Stream := TFileStream.create ('c:\test.dat', fmCreate);
  Int := 10;
  Str := 'test stream';
  Stream.Write (Int, SizeOf (Integer));
  Stream.Write (Str [1], Length (Str));
  Stream.free;
End;

```

在上述代码中用 Write 方法来完成数据的存储操作。除 Write 方法外, 还可用 WriteBuffer 和 ReadBuffer 将数据存入流中或从流中读取数据, 其功能与 Write 和 Read 相同。前面代码中的 Write 语句若用 WriteBuffer, 则相应语句为:

```

Stream.WriteBuffer (Int, Sizeof (integer));
Stream.WriteBuffer (Str [1], Length (str));

```

在 Stream 中可存入各种数据。在一个存有各种混合数据类型的流中, 程序员必须为流中的数据制定相关的数据结构以便能从流中正确读出相关的数据。例如在上述代码中, 我们在流中存入了一个整数和一个字符串。若只存取这两个数, 程序还是可正确读出的; 若在字符串后面还存了一个整数、字符串等任一数据, 则我们就无法正确读出了。在实际编程中, 往往要在每个数据前加一类型标识, 在字符串前要存入字符串的长度, 这样就可以正确读出所有的数据了。当然, 对于不同的具体应用, 可能还有其他更简单的方法使得数据能正确恢复。

除可用 Read 和 Write 进行流数据的读写操作外, TStream 还引入了与组件和文件管理器相关的存取方法, 这些方法能存取现有窗口的组件及其祖先窗口类中的组件。这些方法是由初始化组件流的全局过程自动调用的, 当然也可以直接调用这些方法进行流的初始化处理工作。有

一点需要说明的是, 组件的流化处理需要涉及到两个对象, 一个是组件对象, 另一个是文件管理器。组件对象是以参数形式传给相关方法的, 而文件管理器对象是由 TStream 类自动创建的, 并且自动与 TStream 关联在一起。TStream 是抽象类而不能直接实例化; 对于用组件流化操作的 TStream 的派生类则是由全局函数 ReadComponentResFile 和 WriteComponentResFile 自动创建的。ReadComponentResFile 和 WriteComponentResFile 分别用于从资源文件中读出对象和将对象保存到资源文件中。

10.3.2 TFileStream 与 TMemString

TFileStream 和 TMemString 是两个最常用的 TStream 派生类, TFileStream 为文件的操作提供了基于类的操作, 而 TMemString 是基于内存的流化存储操作类。用 TFileStream 对文件操作与用传统的文件操作相比具有许多优越性。一方面在存储操作时更简单, 特别是对有混合数据类型的操作时更是如此; 另一方面它便于与其他的数据流化操作相接口, 且易于在面向对象的编程中使用和扩展。例如, 要实现两个文件的复制, 用 TFileStream 来操作就变得相当简单了, 相应的代码如下:

```
Procedure CopyFile (const FromFileName, ToFileName: String);
Var
  FromStream, ToStream: TFileStream;
Begin
  FromStream := TFileStream.Create (FromFileName, fmOpenRead);
  ToStream := TFileStream.Create (ToFileName, fmOpenWrite or fmCreate);
  ToStream.CopyFrom (FromStream, FromStream.Size);
  ToStream.Free;
  FromStream.Free;
End;
```

对于要进行加密的数据而言, 在程序运行时所处理的数据往往要存在内存中, 最后经加密后才会存到文件中, 这样做的原因有两个: 程序在运行时所输入或生成的原始数据放在内存流面不是在临时文件中是为了保证数据的安全; 同时运行的效率也较高, 处理速度更快。示例程序 10-7 是一个文件加密函数, 这是用异或进行简单加密的例子。

示例程序 10-7 文件加密函数

```
Function Encode (Source: TMemStream; Const SaveFileName: String;
  const Key: Word): Boolean;
Var
  FileStream: TFileStream;
  Tmp: Word;
  Count: integer;
Begin
  If Assigned (Source) then
  Begin
    FileStream := TFileStream.Create (SaveFileName, fmCreate);
    Try
      Count := 0;
      While Count < Source.Size do
      Begin
```

```

        Source.Read (Tmp, 2);
        Tmp := Tmp Xor Key;
        FileStream.Write (Tmp, 2);
        Inc (Count, 2);
    End;
    Result := True;
Finally
    FileStream.Free;
    Result := false;
End;
End
Else Result := false;;
End;

```

在上述代码中，Source 是 TMemStream 流，存放有要存盘的原始数据，而 Key 是密钥。该过程只能处理 TMemStream 中存储的数据为偶数字节的数据；若是奇数字节，则会出错。若要改为能处理奇数字节的，只要循环的控制加以简单的修改即可完成。

10.3.3 TCompressionStream 和 TDecompressionStream

TCompressionStream 和 TDecompressionStream 与其他流对象不同，它们分别用于数据的压缩和解压（采用 ZIP 压缩算法）。该类的代码并未完全公开，一些与 ZIP 算法核心相关的代码以 *.OBJ 的方式提供，并且压缩后的文件与现在流行的 *.ZIP 文件不同。不同点在于，*.ZIP 保存有原来的文件名、路径等信息，因此可实现多个文件压缩为一个文件，如 WinZip 和 Rar 工具等，这些文件的结构基本相同。而 TCompressionStream 和 TDecompressionStream 只完成了数据压缩算法部分。因此只能对数据进行压缩和解压，而对于源文件名和文件路径等这些非压缩信息则必须由外围程序自行处理。使用 TCompressionStream 和 TDecompressionStream 处理单个文件的压缩和解压是十分方便和简单的。若要完成将文件处理为流行的 *.ZIP 格式，则还必须编写外围的一些代码，这些代码与具体的文件格式是紧密相关的，只要了解了文件格式，用 TCompressionStream 和 TDecompressionStream 编写一个 ZIP 文件的压缩解压工具还是很简单的。

由于 TCompressionStream 和 TDecompressionStream 只完成数据的压缩解压算法，而与数据的具体存储媒介无关，因此需要与其他的 TStream 派生类相配合才能完成数据的压缩解压和存取功能，而 TStream 的派生类又随存储媒介等的不同有许多派生类，这些派生类与 TCompressionStream 和 TDecompressionStream 相配合将有很高的应用意义。如：TCompressionStream 和 TDecompressionStream 与 TFileStream 相配合可完成文件的压缩和解压缩；与 TBlobStream 配合可实现对大型数据库中 BLOB 字段数据的压缩和解压缩；与 TWinSocketStream 配合可实现网上传输数据的压缩和解压缩，如电子邮件和网上聊天内容等。若再与某一具体的加密算法相结合，将使网上传输的数据具有相当的安全性。下面我们以简单文件压缩解压为例说明 TCompressionStream 和 TDecompressionStream 的应用。

示例程序可以完成一个文件的压缩和解压，运行界面如图 10-7 所示。压缩时将文件名与文件内容一起压缩，解压时将解压文件覆盖原被压缩的文件。具体代码如示例程序 10-8 所示。



图 10-7 压缩解压流示例程序运行界面

示例程序 10-8 压缩解压流示例程序

```

unit CompressMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Zlib, zlibConst, FileCtrl, StdCtrls;

type
  TForm1 = class (TForm)
    BtnCompress: TButton;
    BtnUncompress: TButton;
    DriveComboBox: TDriveComboBox;
    DirectoryListBox: TDirectoryListBox;
    FileListBox: TFileListBox;
    procedure BtnCompressClick (Sender: TObject);
    procedure BtnUncompressClick (Sender: TObject);
  private
    }Private declarations {
  public
    }Public declarations {
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.BtnCompressClick (Sender: TObject);
var
  FileName: string;
  LStr: string;
  SrcStream, DestStream: TFileStream;
  Int: Integer;

```

```

    Compress: TCompressionStream;
begin
    FileName: = ExtractFileName (FileListBox.FileName);
    if Length (FileName) > 0 then
    begin
        try
            LStr: = FileListBox.FileName;
            SrcStream: = TFileStream.Create (LStr, fmOpenRead);
            LStr: = LStr + '_zip';
            DestStream: = TFileStream.Create (LStr, fmCreate);
            DestStream.Position: = 0;
            Compress: = TCompressionStream.Create (clDefault, DestStream);
            LStr: = FileListBox.FileName;
            Int: = Length (LStr);
            Compress.Write (Int, sizeof (Integer) );
            Compress.Write (LStr [1], Length (LStr) );
            Int: = SrcStream.Size;
            Compress.Write (Int, sizeof (Integer) );
            Compress.CopyFrom (SrcStream, int );
        finally
            Compress.Free;
            DestStream.Free;
            SrcStream.Free;
            FileListBox.Refresh;
        end;
    end;
end;

procedure TForm1.BtnUncompressClick (Sender: TObject);
var
    FileName: string;
    LStr: string;
    SrcStream, DestStream: TFileStream;
    Int: Integer;
    Compress: TDecompressionStream;
begin
    try
        SrcStream: = TFileStream.Create (FileListBox.FileName, fmOpenRead);
        SrcStream.Position: = 0;
        Compress: = TDecompressionStream.Create (SrcStream);
        Compress.Read (Int, sizeof (Integer) );
        SetLength (FileName, Int);
        Compress.Read (FileName [1], Int);
        DestStream: = TFileStream.Create ('test'{FileName}, fmCreate);
        Compress.Read (int, sizeof (integer) );
        DestStream.CopyFrom (Compress, Int);
    finally
        Compress.Free;
        DestStream.Free;
        SrcStream.Free;
        FileListBox.Refresh;
    end;
end;

end.

```

注意 要使用 TCompressionStream 和 TDecompressionStream, 需要在单元接口部分添加 ZLib 单元的引用。(如果使用 Delphi 5, ZLib.pas 可在 Info \ Extras \ Zlib 目录下找到。)

10.4 VCL 的可视化工作机制

VCL 中有大量的组件可以通过 Delphi 主界面上的组件面板进行拖放使用, 协助程序员完成应用程序的可视化设计。Delphi 真正做到了将面向对象开发 (OOD) 和快速应用开发 (RAD) 的统一, 体现了其他面向对象语言所不具备的优势。在程序的设计期, Delphi 将用户的窗口及窗口中组件的属性都存放在 *.dfm 文件中, 该文件为文本文件, 可用任一文本编辑器查看和修改设置。Delphi 在打开一个窗口时, 读取窗口的 DFM 文件, 根据文件中的数据调用各组件的构造函数创建组件, 并设置有关的属性来实现可视化设计。Delphi 还提供了 Object Inspector, 可视化地协助用户完成对象属性值的设置。

Delphi 为 VCL 提供了可视化的工作机制, 其中主要使用了流技术和 RTTI 技术。本节中我们将介绍这方面的内容。

10.4.1 TFiler 类、TReader 类和 TWriter 类

若要理解 Delphi 可视化的工作机制就必须理解 TFiler 类及其派生类。TFiler 是一抽象基类, 它有两个派生类, 即 TReader 和 TWrite 两个类。TFiler 主要完成组件读取和保存组件及组件的属性, 如在设计期将窗口及其内部组件的属性存放到 DFM 文件中。

TFiler 的对象需要与具体的流对象和要存取的对象相配合才能正常工作。Delphi 的可视化操作就是通过 TFiler 的派生类 TReader 和 TWriter 来完成对象数据的读取和重构。TFiler 在 Classes 单元中声明如下:

```
TFiler = class (TObject)
protected
  procedure SetRoot (Value: TComponent); virtual;
public
  constructor Create (Stream: TStream; BufSize: Integer);
  destructor Destroy; override;
  procedure DefineProperty (const Name: string;
    ReadData: TReaderProc; WriteData: TWriterProc;
    HasData: Boolean); virtual; abstract;
  procedure DefineBinaryProperty (const Name: string;
    ReadData, WriteData: TStreamProc;
    HasData: Boolean); virtual; abstract;
  procedure FlushBuffer; virtual; abstract;
  property Root: TComponent read FRoot write SetRoot;
  property LookupRoot: TComponent read FLookupRoot;
  property Ancestor: TPersistent read FAncestor write FAncestor;
  property IgnoreChildren: Boolean read FIgnoreChildren
    write FIgnoreChildren;
end;
```

TFiler 创建时需要传入两个参数: Stream 参数传入要操作的流对象, 流对象必须预先创建

好, 而 BufSize 传入使用的缓冲区大小。在 TFile 的方法中, DefineProperty 和 DefineBinaryProperty 两个方法是它的核心, 这两个方法使数据的读取像读写属性操作一样方便简单。两个方法的区别在于, 后者主要用于读写一些以二进制为主的数据, 如图形数据或声音数据等, 并通过参数传入实际完成读写数据的过程。

TReader 和 TWriter 都是 TFile 的派生类, 前者用于数据的读取, 而后者用于数据的存储。两者除实现了 TFile 所声明的各个虚方法和抽象类方法外, 还针对各种基本的数据类型和组件属性数据的读写声明了对应的方法, 如 ReadDate、WriteFloat 等。由于按各种数据定义有相应的存取方法, 因此, 对一些数据结构的存取操作将变得十分简单方便, 而不用像流操作那样需要组织好文件的结构, 否则有可能导致读出的数据与写入的数据不相同。下面我们以一个简单的例子来说明。

```
Type
  TCircle = record
    Center: TPoint;
    Radius: Double;
    Color: TColor;
    Label: String;
  End;

TCircleAry = Array of TCircle;

Procedure WriteCircle (const StoreFileName: String;
                      Const CircleAry: TCircleAry);
Var
  Stream: TFileStream;
  Writer: TWriter;
  I: integer;
  //sub Procedure
  Procedure WriteACircle (Circle: TCircle);
  Begin
    Writer.WriteInteger (Circle.Center.X);
    Writer.WriteInteger (Circle.Center.Y);
    Writer.WriteFloat (Circle.Radius);
    Writer.WriteInteger (Circle.Color);
    Writer.WriteString (Circle.Label);
  End;
  // sub Procedure end
Begin
  Stream := TFileStream.Create (StoreFileName, fmCreate);
  Writer := TWriter.Create (Stream, 100);
  Writer.WriteInteger (Length (CircleAry));
  For i := 0 to Length (CircleAry) - 1 do WriteACircle (CircleAry [i]);
  Writer.FlushBuffer;
  Writer.free;
  Stream.Free;
End;

Procedure ReadCircle (const FileName: String): TCircleAry;
Var
  Stream: TFileStream;
```

```
Reader: TReader;  
Len, i: integer;  
//sub Function  
Function ReadACircle: TCircle;  
Begin  
  Result.Center.X: = Reader.ReadInteger;  
  Result.Center.Y: = Reader.ReadInteger;  
  Result.Radius: = Reader.ReadFloat;  
  Result.Color: = Reader.ReadInteger;  
  Result.Label: = Reader.ReadStr;  
End;  
//sub Function end  
Begin  
  Stream: = TFileStream.Create (FileName, fmOpenRead);  
  Reader: = TReader.Create (Stream, 100);  
  Len: = Reader.ReadInteger;  
  SetLength (Result, Len);  
  For i: = 0 to Len-1 do Result [i]: = ReadACircle;  
  Reader.Free;  
  Stream.Free;  
End;
```

从上面的例子中我们可以看出, 要存取一些具有复杂数据结构的数据时, 用 TReader 和 TWriter 类进行操作是十分方便省事的。该例子只是用过程来处理数据的存取操作; 如果是一个非常复杂的数据 (如矢量图形数据的存储), 甚至可设计 TReader 和 TWriter 的派生类。这样便于在实际开发中使用; 不仅程序的可读性好, 而且随着数据结构的变化, 应用程序的维护工作也变得相对简单。

10.4.2 TStream 和组件属性的存取

TStream 类在 Delphi 的可视化设计中扮演了一个十分重要的角色, 在程序的设计期, 窗口及窗口中组件的属性都存在 *.dfm 文件中, 该文件为文本文件, 可用任一文本编辑器查看和修改设置。Delphi 在打开一个窗口时, 就是读取窗口的 DFM 文件, 根据文件中的数据调用各组件的构造函数创建组件, 并设置有关的属性来实现可视化设计的。若要看可视化窗口的有关属性内容, 只要右击鼠标选 View As Text, 则 Delphi 以文本的方式显示窗口及其组件对象的有关属性设置值。对于编译后的程序, 原来存在 DFM 文件中的有关属性被编译程序编译到程序的资源中, 在程序运行时由相关过程将其从资源中读出而创建有关组件, 并根据资源中的数据设置组件的属性, 以完成窗口的重构工作。

为了能将窗口及其组件的属性存到数据流中, 且能从数据流中读出有关的属性, TStream 对象为此提供了两个方法, WriteComponentRes 用于将属性存到流中, 而 ReadComponentRes 的作用刚好相反。WriteComponentRes 和 ReadComponentRes 更适合用于与 DFM 文件相关的操作, 因此在设计期可用这两个函数。如果资源是存在于内存中, 则用 WriteComponent 和 ReadComponent 两个方法会更合适 (比如在程序的运行期)。

当窗口及其组件的属性用 WriteComponent 方法存到内存流中时, 由于数据在流中以二进制的方式存在, 因此如何显示出来以查看就成为一个问题了。当我们右击鼠标选 View As Text 菜

单显示窗口的属性时，Delphi 并不是将 DFM 文件读出来显示给我们看，因为此时可视化窗口的设置由于用户还未存盘而使得所见的内容与 DFM 所存的内容不一致，Delphi 用 WriteComponent 将设置存到内存流中，然后再转换为可视的字符串将有关设置在窗口中显示出来。为了将流中以二进制存储的属性转换为可视的文本或文本属性要转换为可视的二进制流，Delphi 在 Class 单元中声明了四个过程来完成数据的转换。如：

```
procedure ObjectBinaryToText (Input, Output: TStream); overload;  
procedure ObjectBinaryToText (Input, Output: TStream; var OriginalFormat:  
TStreamOriginalFormat); overload;  
procedure ObjectTextToBinary (Input, Output: TStream); overload;  
procedure ObjectTextToBinary (Input, Output: TStream; var OriginalFormat:  
TStreamOriginalFormat); overload;  
procedure ObjectResourceToText (Input, Output: TStream); overload;  
procedure ObjectResourceToText (Input, Output: TStream; var OriginalFormat:  
TStreamOriginalFormat); overload;  
procedure ObjectTextToResource (Input, Output: TStream); overload;  
procedure ObjectTextToResource (Input, Output: TStream; var OriginalFormat:  
TStreamOriginalFormat); overload;
```

为了说明流在 Delphi 可视化设计中的核心作用，下面的例子能说明 Delphi 可视化设计实现的思路和方法。例子由两个简单的程序组成，即由 WriteComponentPrj 和 ReadComponentPrj 组成。WriteComponentPrj 中设有一个 TMemo，我们能在其中输入一些文本，最后可按下按钮将 Memo 组件的属性全部存到一个文本文件中（类似 DFM 文件）；而 ReadComponentPrj 程序能从保存的文本文件中恢复一个与在 WriteComponentPrj 中设置一样的 TMemo 类。图 10-8 为 WriteComponentPrj 的窗口，而图 10-9 为 ReadComponentPrj 程序恢复的窗口。

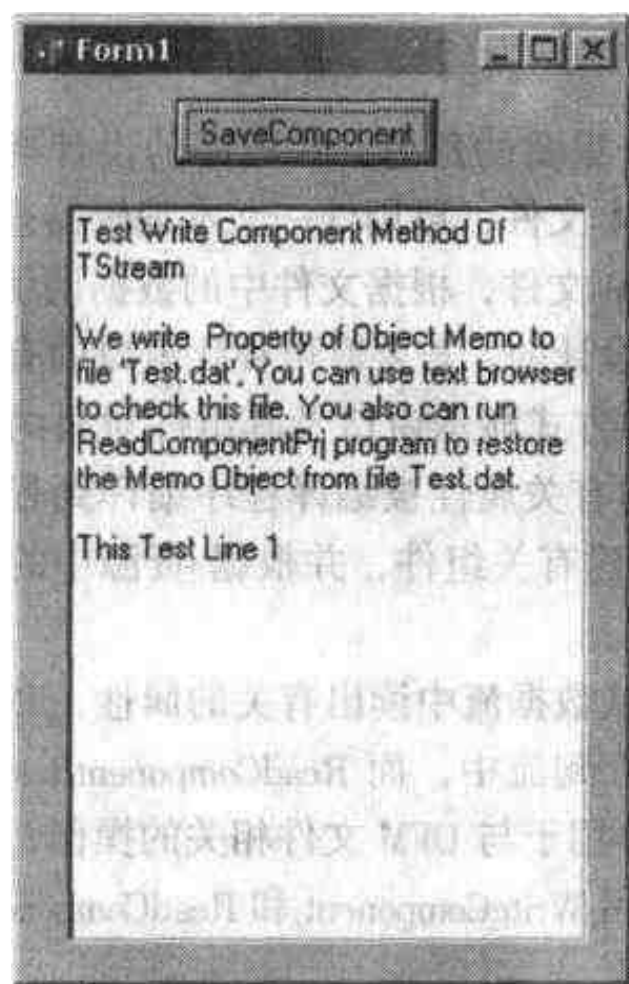


图 10-8 WriteComponentPrj 的运行窗口

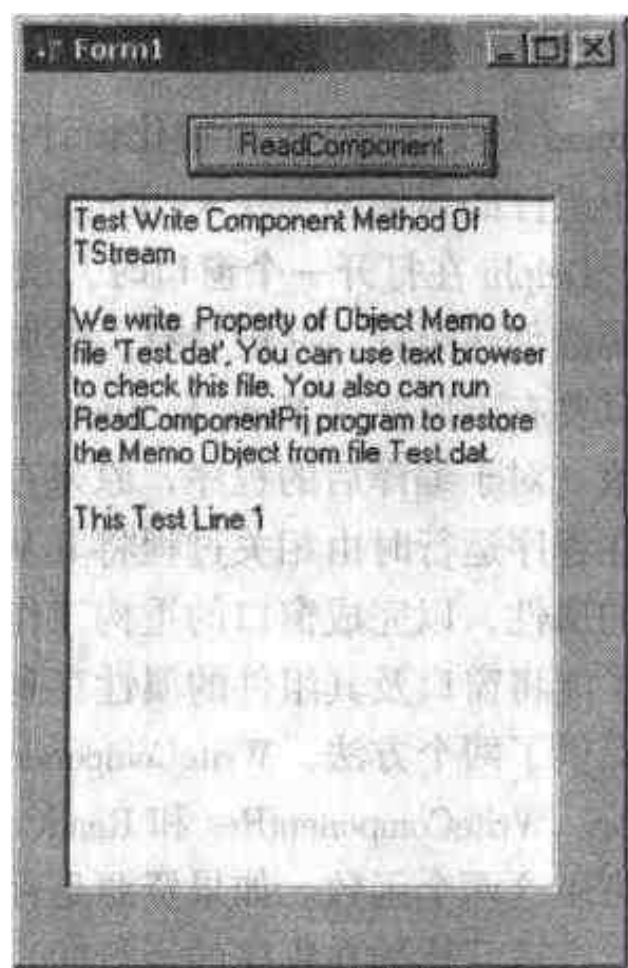


图 10-9 ReadComponentPrj 恢复的窗口

下面是 WriteComponentPrj 和 ReadComponentPrj 程序的源代码, 及 WriteComponentPrj 所产生的类似 DFM 的用于保存 Tmemo 组件属性的数据文件。

```

object SaveComponent: TBitBtn
  Left = 56
  Top = 8
  Width = 97
  Height = 25
  Caption = 'SaveComponent'
  TabOrder = 0
  OnClick = SaveComponentClick
end
object Memo: TMemo
  Left = 16
  Top = 48
  Width = 193
  Height = 273
  ImeName = #20116#31508#22411#30721
  Lines.Strings = (
    'Memo')
  TabOrder = 1
end

unit ReadComponentMain;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;
type
  TForm1 = class (TForm)
    ReadComponent: TButton;
    procedure ReadComponentClick (Sender: TObject);
  private
    {Private declarations}
    Memo: TMemo;
  public
    {Public declarations}
  end;
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.ReadComponentClick (Sender: TObject);
var
  MemStream: TMemoryStream;
  FileStream: TFileStream;
begin
  FileStream := TFileStream.Create ('Test.dat', fmOpenread);
  FileStream.Position := 0;
  MemStream := TMemoryStream.Create;

```

```

ObjectTextToBinary (FileStream, MemStream);
Memo := TMemo.Create (self);
Memo.Parent := Self;
Memstream.Position := 0;
Memo := TMemo (MemStream.ReadComponent (MEMO));
FileStream.Free;
MemStream.Free;
end;
end.

```

Test.dat 数据文件如下:

```

object Memo: TMemo
  Left = 16
  Top = 48
  Width = 193
  Height = 273
  Name = #20116#31508#22411#30721
  Lines.Strings = (
    'Test Write Component Method Of '
    'TStream'
    ''
    'We write Property of Object Memo to '
    'file'#39'Test.dat'#39', You can use text browser '
    'to check this file. You also can run '
    'ReadComponentPrj program to restore '
    'the Memo Object from file Test.dat.'
    ''
    'This Test Line 1'
    '')
  TabOrder = 1
end

```

Delphi 不仅可以运行期组件的属性用流化技术存到内存、文件中, 还可将运行着的程序的组件或窗体的属性从程序的资源文件中读出, 同时也可对运行期程序的资源进行设置和修改。利用这一技术, 可将任意程序的窗口及其组件的属性从资源中读出, 同理也可对其进行设置。有兴趣的读者可用下面介绍的方法来实现。要从程序的资源中读出窗口及组件的有关属性, 要用到 TResourceStream。该对象是专门为操作运行着的程序资源或 DLL 资源而设计的, 全局函数 ReadComponentRes 和 WriteComponentRes 就是利用 TResourceStream 对程序或 DLL 的资源进行读取和修改工作的。TResourceStream 的构造函数与其他流类不同, 它特别声明为:

```

constructor Create (Instance: THandle; const ResName: string; ResType: PChar);

```

Instance 参数传入要操作的程序的实例句柄, 参数 ResName 传入要操作的资源名称, 而 ResType 传入资源的类型 (如快捷键、字体字形、对话框或菜单等等)。其他方法与前面介绍的流类基本相同, 因此就不一一介绍了, 具体请参阅 Delphi 的在线帮助。

前面例子介绍的关于组件属性的保存和组件属性的恢复实际上还有更简单的方法来完成, 我们之所以用前面的方法, 主要是想使读者对流的功能和 Delphi 可视化设计实现的机理有更深入的理解。对于 Memo 组件的保存只要用一条语句即可完成, 具体为:

```

WriteComponentResFile ('Test.dat', Memo);

```

相应的恢复也十分简单,如:

```
Memo := ReadComponentResFile ('Test.dat', nil) as TMemo;
```

函数 WriteComponentResFile 和 ReadComponentResFile 是 Delphi 的全局函数。

10.4.3 Object Inspector 的工作原理

为什么 Object Inspector 能正确显示对象属性的设置值呢?关于这个问题,目前还不是非常清楚,这主要是 Delphi 没有提供这方面的文档资料。由于没有正式公开相关的文档,因此, Borland 公司在 Delphi 的版本升级时可根据需要改变所需的数据结构和函数,这样不会为版本兼容而增添负担。虽然没有公开的文档,但有一点是肯定的, Object Inspector 是利用 RTTI 来实现的,相关的数据结构、函数和过程定义在 TypInfo 单元中。Object Inspector 就是调用单元的有关函数和过程来实现对象属性值的获取和设置的。对于 TypInfo 单元, Delphi 并未提供太多的说明文档,而且从 Delphi 1~7 版看,这一单元的变化不是很大,只是数据结构在版本升级时有些变化,而函数和过程基本没变。但从 Delphi 5 开始, Delphi 为 TypInfo 单元增加了一些函数和过程的注释。打开 TypInfo 单元就会发现, Delphi 在该单元中声明了大量与对象属性、过程等设置和取值相关的函数和过程。关于 TypInfo 单元,在这里我不想过多地介绍其中的函数,而只介绍有助于理解 Object Inspector 工作原理的几个函数,这几个函数为: GetPropList、GetPropValue、SetPropValue。下面我们以一个访问对象属性的示例程序 10-9 来说明。该程序可以访问属性名称和读写属性值,运行界面如图 10-10 所示,它模拟 Object Inspector 的工作机制。

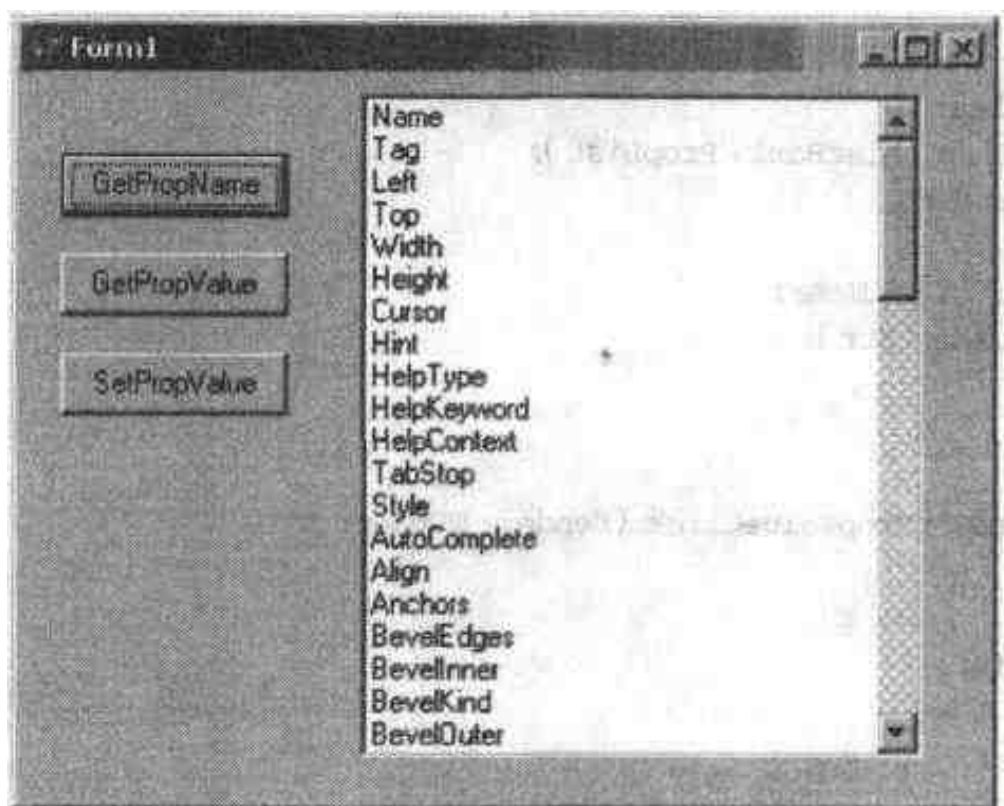


图 10-10 访问对象属性程序的运行界面

示例程序 10-9 访问对象属性

```
unit Main;  
interface  
uses
```

```

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, TypInfo, StdCtrls;
type
  TForm1 = class (TForm)
    ListBox1: TListBox;
    GetPropName: TButton;
    BtnGetPropValue: TButton;
    BtnSetPropValue: TButton;
    procedure GetPropNameClick (Sender: TObject);
    procedure BtnGetPropValueClick (Sender: TObject);
    procedure BtnSetPropValueClick (Sender: TObject);
  private
    {Private declarations }
  public
    {Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.GetPropNameClick (Sender: TObject);
Var
  Count: integer;
  PropList: PPropList;
  i: integer;
  Str: String;
begin
  ListBox1.Items.Clear;
  Count: = GetPropList (ListBox1, PropList);
  For i: = 0 to Count-1 do
  begin
    Str: = PropList^ [i] ^.Name;
    ListBox1.Items.Add (Str);
  end;
end;

procedure TForm1.BtnGetPropValueClick (Sender: TObject);
Var
  Count: integer;
  PropList: PPropList;
  i: integer;
  Name, Str: String;
  Value: Variant;
  Int: integer;
begin
  ListBox1.Items.Clear;
  Count: = GetPropList (ListBox1, PropList);
  For i: = 0 to Count-1 do
  begin
    Name: = PropList^ [i] ^.Name;
    Value: = GetPropValue (ListBox1, Name);

```



```

    Case VarType (Value) of
      VarString:
        begin
          Str: = Format ('%s = %s', [Name, Value] );
          ListBox1.Items.Add (Str);
        end;
      VarInteger:
        begin
          int: = Value;
          Str: = Format ('%s = %d', [Name, Int] );
          ListBox1.Items.Add (Str);
        end;
    end;
  end;
end;

procedure TForm1.BtnSetPropValueClick (Sender; TObject);
Var
  i: integer;
begin
  for i: = 0 to ComponentCount-1 do
    begin
      if not (Components [i] is TListBox) then
        SetPropValue (Components [i], 'Caption', 'Test SetPropValue');
      end;
    end;
  end;

end.

```

在示例程序 10-9 中，为了利用 GetPropList 函数取得 ListBox1 所有的 published 属性和事件，首先调用 GetPropList 得到对象 published 属性和事件的列表，然后将属性名显示在 ListBox1 中。GetPropList 函数有多个重载版本，其中之一声明为：

```
function GetPropList (AObject; TObject; out PropList: PPropList): Integer;
```

该函数由 PropList 参数取得对象的 published 属性和事件的有关信息的列表，函数返回列表的项数。PPropList 声明如下：

```

PPropInfo = ^TPropInfo;
TPropInfo = packed record
  PropType: PTypeInfo;
  GetProc: Pointer;
  SetProc: Pointer;
  StoredProc: Pointer;
  Index: Integer;
  Default: Longint;
  NameIndex: SmallInt;
  Name: ShortString;
end;
PPropList = ^TPropList;
TPropList = array [0..16379] of PPropInfo;

```

从该数据类型可知，PropList 参数不仅返回属性名，还返回了属性的缺省值、存取属性的

过程指针和属性值，以及将值存到 DFM 文件所用的过程指针等。

GetPropValue 和 SetPropValue 的调用相对比较简单，在此就不多介绍了。虽然 TypInfo 单元声明了许多功能强大的底层过程和函数，但在实际编程中要注意，尽量不要使用该单元的函数和过程（因为该单元没有版本兼容的保证，且主要是为 Object Inspector 工具提供的）。使用该单元的函数一方面会降低程序的可移植性，使程序变复杂和降低程序的运行效率；另一方面也会由于程序没经过类型检查而增加了隐藏 bug 的可能性。如果一定要调用 TypInfo 单元的函数或过程，也应使用最新版本的单元来提高程序的兼容性。

附录 A ModelMaker 使用指南^①

李启元

当前,软件开发人员面对着一个需求快速多变的环境,在更短的周期开发出复杂度更高、功能更强大、性能更优化的软件应用,传统开发工具已不能满足软件开发人员的要求,集成软件开发各阶段所需开发工具的整体解决方案已成为软件开发人员的迫切需求。

Borland 公司从 Delphi 7 开始在其 IDE 环境中集成了 ModelMaker 工具,极大地改善了 IDE 环境导航困难、缺乏辅助设计工具的缺点。作为一种为提高 Delphi 程序员编码效率的“智能”工具,ModelMaker 以面向对象的 CASE 工具形式出现。在使用时你甚至会觉得 ModelMaker 在智能地为你编写代码。集成了 ModelMaker 的 Delphi 7 将建模 (Modeling)、设计模式 (Design Pattern) 融入了开发工具中,结合 Borland Delphi 强大的 RAD 以及 OOP 功能,将提供从设计建模到代码开发过程的强大工具支持。

ModelMaker 与其他传统的建模工具 (Rational Rose 等) 一样支持通过 UML 图形建模,但最大的不同在于其设计和代码生成都是以 Object Pascal 代码的形式表达。而 ModelMaker 与其他 Delphi 代码生成器的不同在于其对复杂模型的全局管理和重组能力。同时 ModelMaker 还支持设计模式。准确地说,ModelMaker 是一个全功能的支持 UML 建模、设计模式、代码生成引擎等特性的 Delphi IDE 扩展代码管理、加工工具。

多数建模工具所绘制的 UML 模型图由于无法利用逆向工程与开发工具同步,在工程的实施过程中开发人员所做的一些改动往往不能迅速反映到设计生成的 UML 图,造成设计模型图与最终代码之间有所差距,从而导致设计生成的 UML 模型图最终只能供参考使用,而不能达到建模与实现的完美统一。ModelMaker 解决这类问题的神奇能力来自于其“活动建模引擎”,该引擎建立起 ModelMaker 内部模型数据,其中存储维护了用户定义的类及类之间的关系、单元文件、UML 图形、文档、帮助等相关元素,并可通过多种视图 (View) 从各个角度来维护管理 ModelMaker 内部模型数据。建模工作完成后,ModelMaker 可以立即协助用户生成可被 Delphi 编译的源代码,后期对模型的任何改变都会立即反映到最终生成的代码中。同时软件开发人员在工程实践中通过代码对设计模型所做的修改也可以通过逆向工程引擎迅速反映到设计模型中,这种从设计到代码实现的无缝转换是目前为止其他任何建模工具所不具备的。

Borland 将 Delphi RAD、OOP 功能与 ModelMaker 强劲的 UML 建模、Object Pascal 代码生成能力结合在一起,协助程序员高效、优质地进行设计与编码工作,以“设计—编码—精炼”方式代替了传统“设计—编码”的工作方式。利用 ModelMaker 可在短时间生成、管理大量的设计模型图,以 UML 图形形式提供设计文档。利用 Delphi IDE 对设计生成的代码进行错误调试,最后利用 ModelMaker 出色的逆向工程能力进行 UML 模型图与实现代码程序之间的逆向同步,从而

^① 本文以 Delphi 7 提供的 ModelMaker 6.2 为蓝本撰写,ModelMaker 其他版本的用法基本与此相同。

完成从设计到开发再到设计的循环开发流程。

A.1 类的设计与代码生成

ModelMaker 作为一款与 Delphi IDE 环境、Object Pascal 语言整合的建模、代码生成工具，为软件开发人员提供了一套融合入 Delphi IDE 环境的从工程项目设计、规划到最终 Object Pascal 实现代码生成的完整的 CASE 工具。ModelMaker 利用“建模引擎”存储了与工程模型相关的各种数据，并最终依据这些数据生成了模型的实现代码。从这里我们可以看出内部模型与最终代码（源文件）之间的关系：内部模型是由“建模引擎”建立的、与工程模型相关的、各种数据的总集；而源文件则是最终的外部文件，对内部模型的修改并不会立刻反映到生成的源文件中，而是由用户在修改完内部模型后，通知 ModelMaker 自动生成外部源文件的。

注意 如果打开了 ModelMaker 的“自动代码生成”开关，对内部模型数据的修改基本上是立刻反映到最终代码中的。

首先让我们对 ModelMaker 有个直观的认识，ModelMaker 的工作界面如图 A-1 所示。

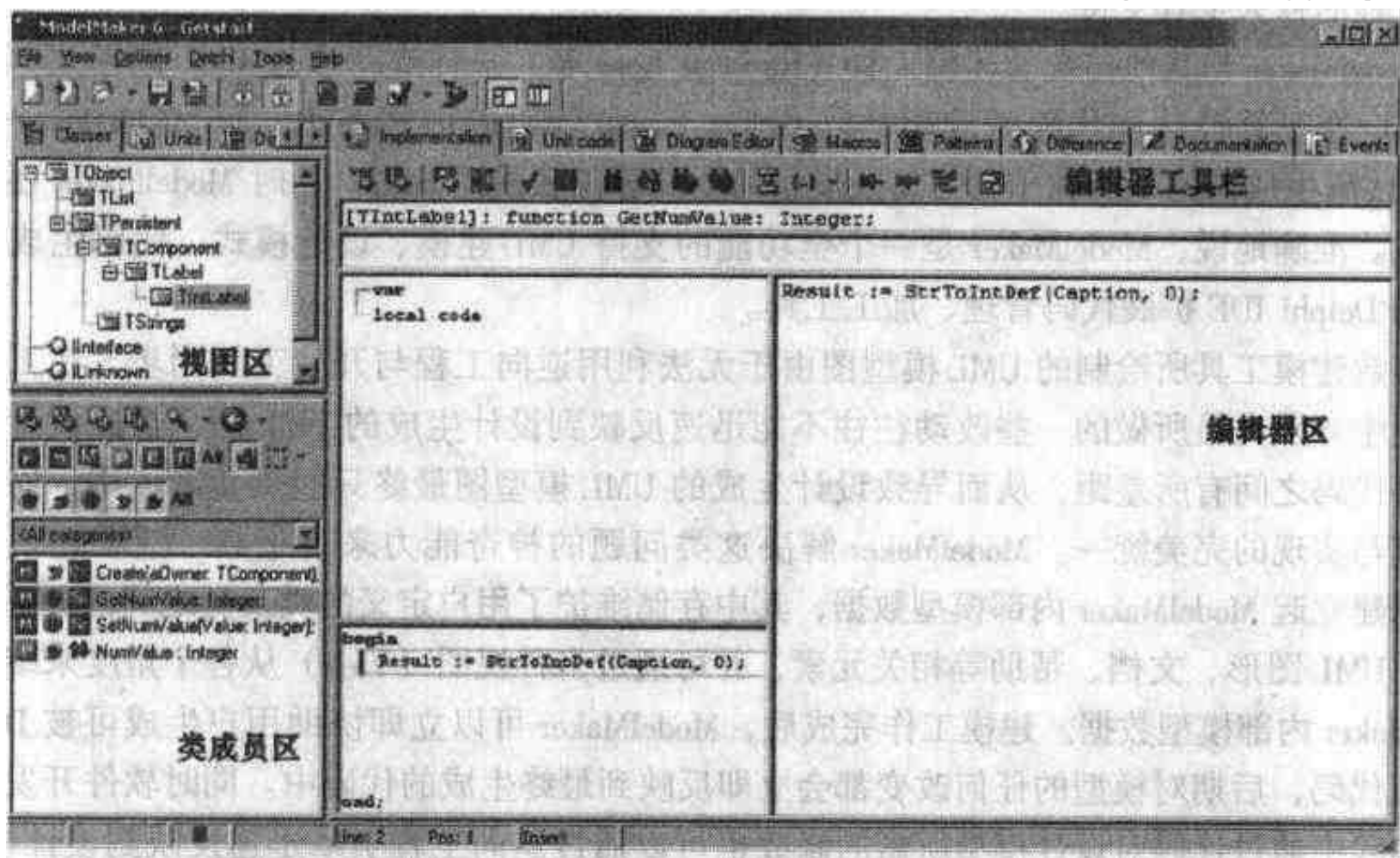


图 A-1 ModelMaker 的工作界面

ModelMaker 将整个工作区域划分为三部分：视图区、类成员编辑区、编辑器区。视图区供设计开发人员在内部模型数据中导航使用，ModelMaker 提供了多种视图（View）从各个不同的角度来观察、修改内部模型数据。类成员区提供了处理当前设计类的各种操作，包括对类的域、属性、方法、函数等成员进行导航、筛选、添加、删除等功能。编辑器区则根据视图区与类成员区的选择自动切换相应的工具条和编辑器供设计编码、绘制 UML 图形等功能，编辑器工具栏相应区域的工具条提供了进行操作的快捷方式。使用过 ModelMaker 的编辑器后就会觉得 Borland Delphi 的 IDE 环境还有太多需要改善的地方。

下面我们通过将一段已编写完成的代码导入 ModelMaker 中来解释 ModelMaker 是如何实现“设计-编码-精炼”的，这种理念必将带来革命性的思维方式，并改变传统的软件开发过程。

注意 在未熟悉 ModelMaker 操作前请首先备份想导入的代码！

在 ModelMaker 中通过主菜单选取 File|New 菜单项或从工具条选择新建按钮创建一个新的建模工程。ModelMaker 的工程项目（“建模引擎”建立的内部模型）中包含了设计过程中开发设计人员所建立的类、类成员的实现代码，UML 建模图形以及建模要素之间的相互关系，并可通过逆向工程引擎实现最终代码源程序与设计模型之间的高度一致。

1. 导入外部源文件

从 Delphi 的早期版本开始，ModelMaker 工具就已存在，发展至今基本上支持几乎所有的 Object Pascal 语言特性，具有极强的 Object Pascal 语法分析引擎，可以引导输入并分析一些编写混乱的代码。但“智能化”软件不是万能的，要取得好的分析效果，最好的方法还是保证导入的代码符合 Object Pascal 语法规则以及一些程序编写规范，这样可以避免许多不应出现的问题。

导入外部源文件，分析其中的语法结构，生成 ModelMaker 内部模型数据，是 ModelMaker 逆向工程引擎的职责。ModelMaker 提供了多种方法导入外部源代码文件，最快捷的方法还是拖拽操作。在 ModelMaker 中按 F4 或选择 Units 标签切换至 Units 视图模式，Units 视图模式会从外部源文件的角度来观察设计、开发所建立的内部模型数据。然后从 Windows 资源管理器中将需导入的代码源文件（PAS 文件）拖入 ModelMaker 主窗体中的 Units 视图中。



注意 导入代码碰到问题时 ModelMaker 会提示用户问题的类型及解决方法。

代码源文件拖入 ModelMaker 后，ModelMaker 的逆向工程引擎立即对代码进行分析，并以树列表的形式将单元和其中所包含的类加入单元列表中，如图 A-2 所示。单元列表中顶层的节点代表刚才导入的单元文件，其下的子节点是在单元源文件中所定义的类、接口和事件，用户双击节点数据可以调用相应的编辑器或对话框来编辑、设置相应的内容。源文件导入时其中的注释也可导入内部模型相应的位置，不过需要在转换时正确设置转换规则。



图 A-2 Units 视图模式

除了拖拽操作外，用户还可以通过 Units 视图单元列表中的上下文菜单项“Import Unit...”（导入单元源文件）或使用主工具条上的按钮来完成外部源文件代码的导入工作：

-  按钮允许用户将外部源文件代码导入一个新的建模工程的内部模型数据中。
-  按钮则将导入的外部源文件代码加入到当前建模工程的内部模型数据中。

代码导入后，按 F3 或选择 Classes 标签切换至 Classes 视图模式。该视图以类之间的层次继承关系显示了内部模型数据之间的相互关系。从外部源文件代码中导入的类以层次继承的树形

显示在 Calsses 列表中, 如图 A-3 所示。注意其中 TObject 类、IInterface 和 IUnknown 接口作为所有类及接口的基础始终显示在列表中。

2. 创建新类

利用 Classes 视图创建新类时, 注意祖先类必须作为内部模型数据的一部分, 否则 ModelMaker 无法正确生成新类的声明部分代码。这似乎表明当用户想生成一个新类时, 首先必须导入其父类, 然后为了导入父类, 则必须导入父类的祖先, 循环往复, 似乎要把整个 VCL 的类继承关系全部导入 ModelMaker 模型中去才可能生成一个新类。这样看上去就太蠢了, ModelMaker 采用一种灵活的方法解决了这个问题, 软件开发人员继承一个类的主要目标是父类所能提供的接口以及子类所需实现的接口, 而并不关心父类的具体实现, 对于父类的祖先就更不关心了, ModelMaker 采用了一种称做“占位符”的标志来简化表示模型

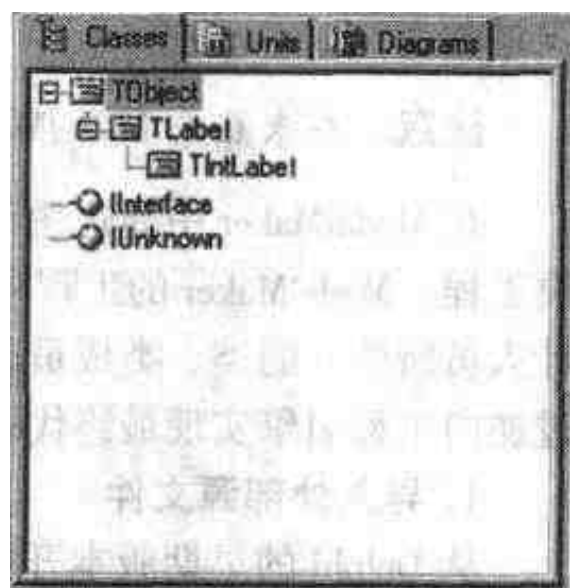


图 A-3 Classes 视图模式

中并不关心其实现的祖先类, 注意图 A-3 中类 TLabel 符号的边框, 虚线类符号表示该类并不是在此模型中实现的, 如始终在模型中出现的 TObject 类, 总以“占位符”类的形式出现。而模型中真实创建的类则以“真实”类的形式出现 (类符号的边框以实线标示, 如图 A-3 中的 TIntLabel 类)。

图 A-3 中的 TLabel 类, 是导入的 TIntLabel 类的父类, 但我们并未导入实现 TLabel 类的单元源文件, 因此 ModelMaker 以“占位符”的形式 (注意 TLabel 前类图标周围的虚线边框) 表示 TLabel 是 TIntLabel 的祖先, 但系统并不关心 TLabel 的具体实现, 至于 TLabel 的祖先类在模型中可以是任何其他类, 内部模型数据并不关心 TLabel 的类层次关系, 根据 Object Pascal 语言的特性, 设置 TLabel 的父类为所有类的祖先 TObject 类, 而实际上 TLabel 类的祖先是 TComponent 类。

如果 TIntLabel 并不是由外部源文件直接导入的, 那么如何利用 ModelMaker 完成添加新类的工作?

首先从 Classes 视图中选中 VCL 类的基类 TObject, 通过选择快捷菜单“Add descendant”或简单按 INS 键可添加一个继承类, 输入继承类的名字, 如 TLabel, 大家会发现此时 TLabel 类以“真实”类的形式出现, 而在模型中并不关心 TLabel 的具体实现, 该类应是模型中的“占位符”类, 如何将其转换为“占位符”类呢? 双击 Class 列表中的 TLabel 类, 在如图 A-4 所示的对话框中勾选 Placeholder 选项, 将 TLabel 类转化为“占位符”类。除了不需将其转换为“占位符”标志, 添加“真实”类 TIntLabel 的方法与上面所述基本一致。

3. 定义类接口与实现

在内部模型中添加、生成了新类, 完成了设计的首要工作类的定义。剩下的工作是为类定义接口, 添加类的属性和方法声明, 这就要用到如图 A-5 所示的 ModelMaker 主界面左下角的类成员视图。类成员视图中按照各个过滤按钮的组合控制 (包括了对显示的成员类型的筛选、访问控制级别的筛选以及用户自定义分类筛选的组合控制) 显示类的复合选择条件的成员 (域、方法、属

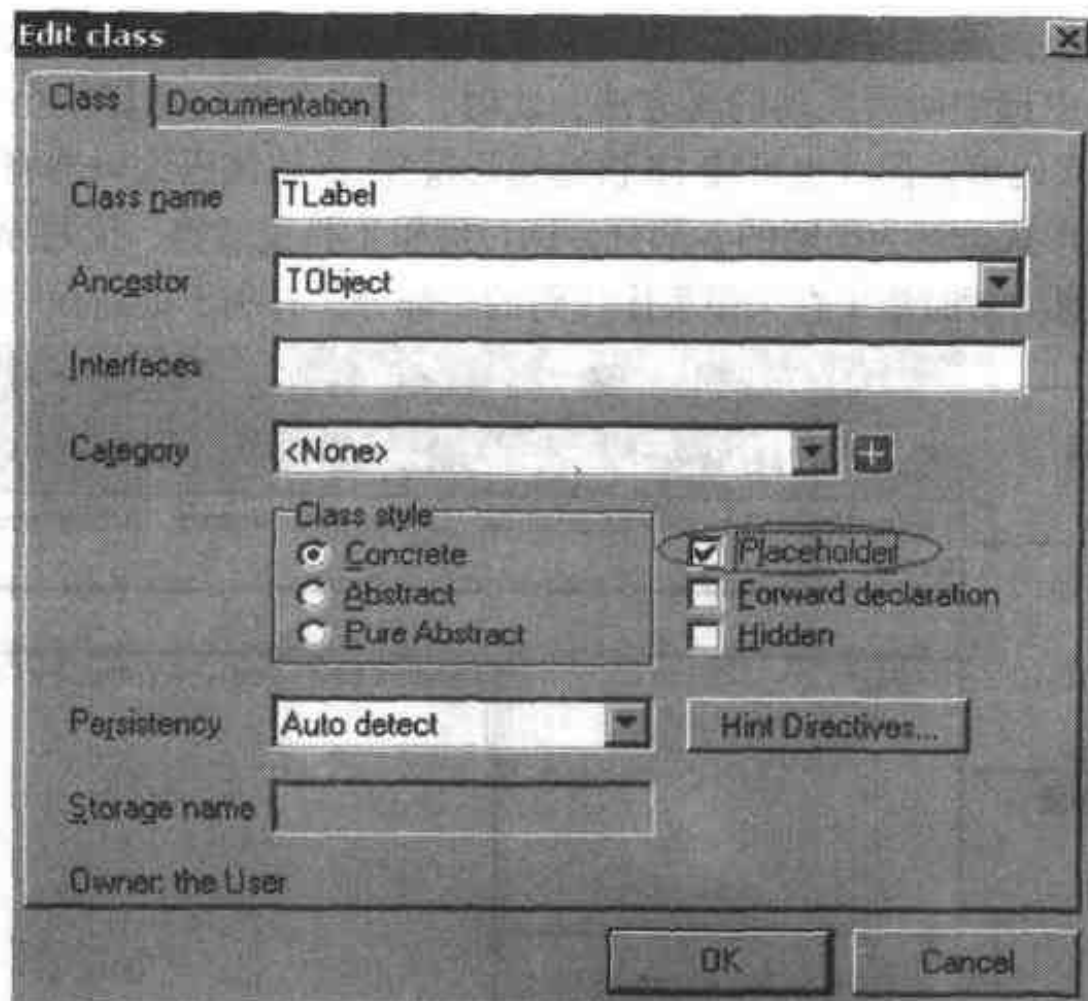


图 A-4 类编辑对话框



图 A-5 类成员视图

性)。因此当有类成员在列表中找到时，注意过滤器集合的选择是否屏蔽了该类成员。

为类添加类成员时，使用类成员视图的“添加类成员工具集”中的按钮可分别添加相应的类成员，如类变量、方法、属性、事件等。此时相应的添加对话框则会弹出供用户处理需添加的类成员，对话框则将原来在 Delphi IDE 环境手工输入的变量、方法、属性、事件等的声明简化为对话框中简单的点选、输入等操作。对话框中以符合 Object Pascal 语言特性的处理方式将变量、方法、属性、事件的定义分离为名称、参数、类型、可见性等相关信息的定义，简化了在 Delphi IDE 环境输入的无序性以及随意性。

类接口定义完成后，就要为刚才定义的类的接口方法编写代码。要完成类接口方法的编码工作，在类成员视图中选中所需实现的类方法，此时，编辑器区自动切换至如图 A-6 所示的函数（或者说是过程）代码编辑器（也可按 F6 快捷键切换）。函数代码编辑器将类方法的实现代码以“节”（Section）的形式分解为串联的各部分，通过编辑工具按钮集、代码导航面板的协助以确保软件开发人员的注意力暂时集中在大段程序代码的一部分，有利于提高程序开发效率。

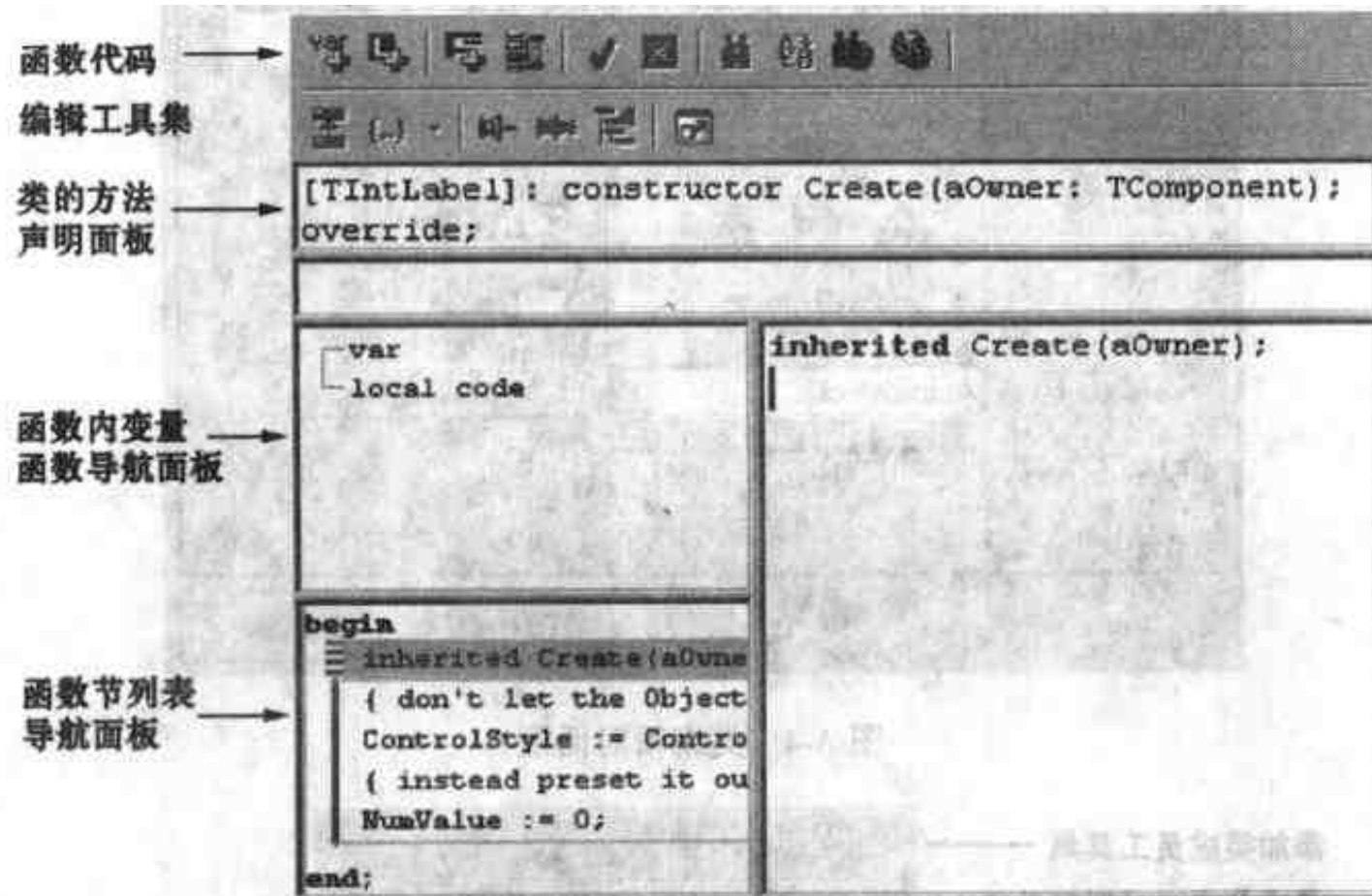


图 A-6 函数代码编辑器

函数代码编辑器透露了 ModelMaker 如何实现单元源代码生成的一部分秘密（其余秘密在后面的代码生成部分介绍）。我们可以看到：类的方法实现部分由局部变量、函数或方法名、在 begin—end 之间由一段或多段代码“节”构成的具体实现代码共同组成。一个代码“节”可以包含任意行 Pascal 代码，函数（方法）头、变量定义、内部方法以及代码“节”共同构成的方法实现代码。利用代码“节”ModelMaker 能够区分实现代码中的特定行。函数代码编辑工具集中提供了“添加节”与“删除节”按钮供程序员对节进行管理。

左下角的函数节列表（Section List）导航面板显示了所有的代码“节”。在一节中包含有大量代码时，内部的代码行会收缩显示，此时所显示的代码段仅仅是一个大略情况。而右侧的代码编辑器则用于编辑所选“节”中所包含的实际代码，编辑器同样可用于编辑实现类的方法所使用的本地过程或函数代码。

注意 你可能注意到在代码“节”列表导航面板中各节左边界由不同的颜色标示：红色线条标示该节内的代码内容由类的祖先继承而来，由其祖先类来维护其中代码的工作。绿色代表该节是由用户创建维护的节。由绿色线条与紫色线条共同组成的标志则表明此节已被收缩显示。

虽然类代码的设计与实现工作到此为止已完成，但并不代表该类就可以在 Delphi 中被使

用, 此时类定义及其实现代码以及各模型要素之间的相互关系仅仅存在于 ModelMaker 所管理的内部模型中。下面我们要完成最重要的工作: 利用 ModelMaker 自动生成 Object Pascal 单元文件。

4. 生成所需单元源代码文件

利用 ModelMaker 建立的模型中所包含的类、类的接口以及类接口的实现代码等最终要转换为可被 Delphi 编译执行的 Object Pascal 单元文件。ModelMaker 内置的正向工程引擎(代码生成引擎)可将内部模型中维护管理的类、类定义、类之间的相互关系及实现代码等转换生成 Delphi 可编译的单元文件。

需要生成单元代码时, 用户切换至单元视图模式下, 并切换为如图 A-7 所示的单元代码编辑器。单元代码编辑器与上节所述的函数代码编辑器虽然都用于编辑实现单元代码, 但函数代码编辑器完成的是类级别上代码的创建与管理, 而单元代码编辑器完成的则是不在类层次上的代码, 如用户自定义类型的声明、元类的声明、单元方法以及变量等的定义, 这些就必须由单元代码编辑器来完成了。

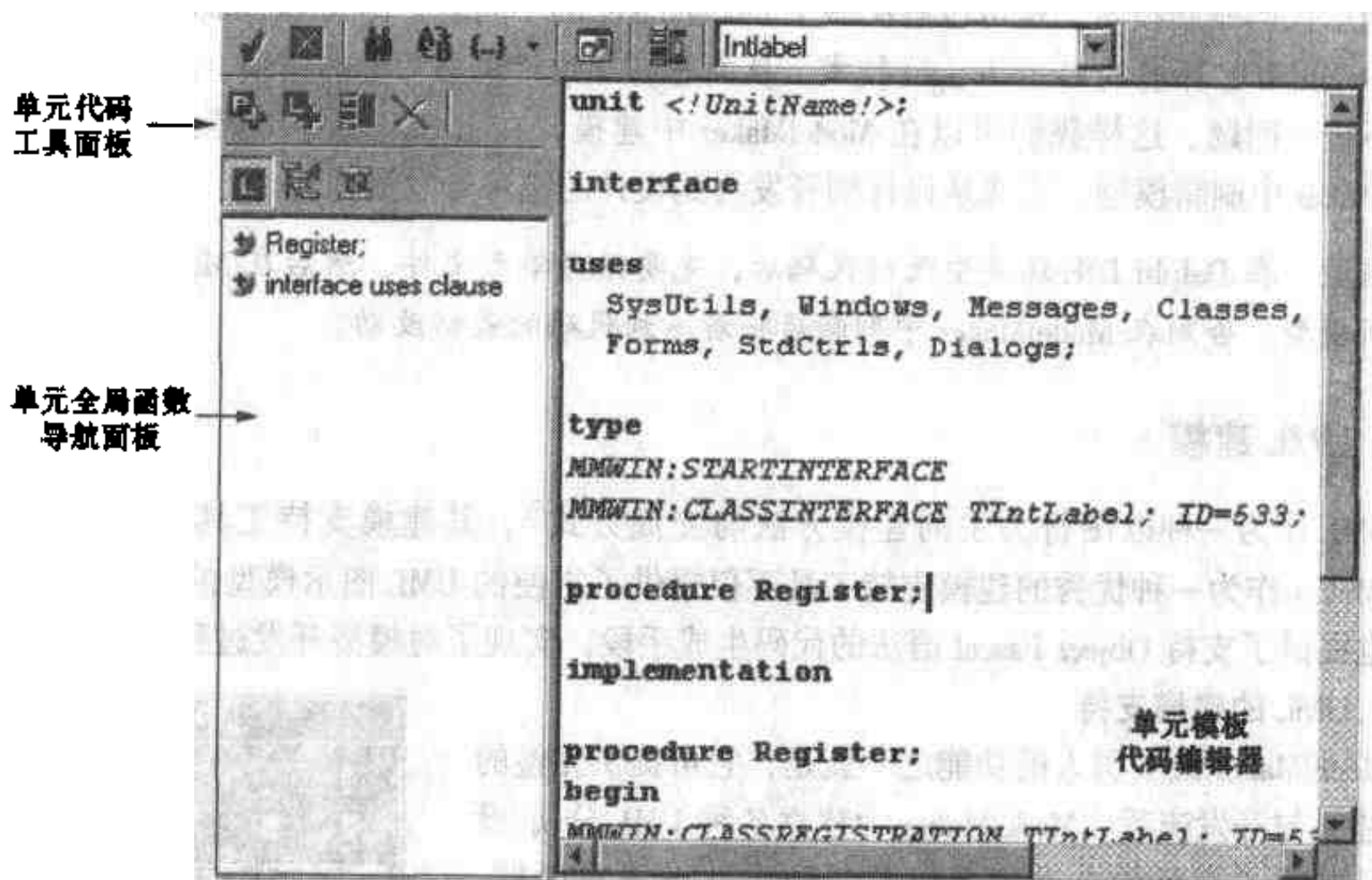


图 A-7 单元代码编辑器

如果单元文件尚未创建, 则首先必须创建一个新的单元 (Unit) 文件, 单击工具栏中的添加单元文件按钮, 完成单元文件属性编辑对话框中关于即将生成的单元源文件的存储位置、文件名称、生成单元文件所需模板以及需要在单元中生成的内部模型中定义的哪些类等内容。此时为内部模型中设计的类提供存储位置的工作已完成。我们可在如图 A-7 所示的单元代码编辑器中看到 ModelMaker 使用的单元模板代码, 其中包含了 ModelMaker 定义的代码生成标志 (类似于 C 语言中的宏标识), 如 “MMWIN: CLASSINTERFACE TIntLabel; ID = 533;” 等标志。这些标志包括了三种类型: 纯文本、宏、代码生成标志。ModelMaker 对这些标志扫描分析后, 以模型中定义的类及其接口定义代换这些标志, 从而生成 Delphi 可编译的单元代码源文件。

注意 软件开发人员可在单元代码编辑器中自由添加或删除相应代码。

单击图 A-2 Units (单元) 视图工具条中的代码生成按钮, ModelMaker 将根据内部模型中所包含的类、类的接口定义以及类的实现代码, 依据所选择的代码生成模板自动生成可被 Delphi 编译的 Object Pascal 单元文件。生成的 Object Pascal 单元文件还必须在 Delphi 环境中编译、查错, 以确保单元的正确性。

当在 Delphi IDE 环境中完成了查错、编译等工作后, 实际的代码文件必然与模型中所设计的类、类的成员、方法实现等有所不同。此时利用 ModelMaker 的“逆向工程引擎”, 可以方便地使模型与实际工程代码保持一致。

单击图 A-2 Units 视图工具条中的代码刷新按钮, ModelMaker 将会根据实际代码文件内容将其中类、接口的定义、成员、方法以及实现代码导入 ModelMaker 的模型中, 更新模型与实际代码不符之处, 以保持建模与实现的完美统一。

利用代码刷新技术, 也可以解决一个 ModelMaker 的小问题: 即 ModelMaker 的代码编辑器不包含 Delphi IDE 环境的 Code Insight 技术, 从而容易导致代码书写上的错误, 造成建模与实现代码的不统一问题。这样我们可以在 ModelMaker 中建模, 在 IDE 环境中编码、测试, 然后再在 ModelMaker 中刷新模型, 完成从设计到开发再到设计的循环开发流程。

注意 在 Delphi IDE 环境中改动代码后, 先要保存单元文件, 然后在 ModelMaker 中刷新模型。否则在 ModelMaker 中刷新后将看不到代码的最新改动。

A.2 UML 建模

UML 作为一种以图符为主的建模方法与交流方式^①, 其建模支持工具就显得非常重要。ModelMaker 作为一种优秀的建模支持工具不仅提供了方便的 UML 图示模型的绘制、更改手段, 同时也提供了支持 Object Pascal 语法的代码生成手段, 实现了对模型开发过程的全面管理。

1. UML 的建模支持

ModelMaker 最吸引人的功能之一就是, 它可提供完整的 UML 建模与开发流程。ModelMaker 为建立各种 UML 分析/设计图形提供了充足的自动工具以及各种属性设置对话框, 让开发人员能够设计完整的模型。ModelMaker 支持 UML 1.3 定义的各个模型, 能够让开发人员使用齐全 (甚至超越 UML 定义) 的 UML 图形来进行设计、分析工作。

ModelMaker 支持以下 UML 模型: 用例图、类图、序列图、状态图、活动图、协作图、实现图、包图、鲁棒图、思维图。ModelMaker 利用模型图视图 (如图 A-8 所示) 以及模型图编辑器来创建以及管理模型中建立的 UML



图 A-8 模型图视图

^① 虽然 UML 也提供了一套 OCL 描述语言, 但在工程实践中主要是使用 UML 的可视化图示模型。

图,以图形化的方式实现对内部模型数据的管理与维护。利用 ModelMaker 的 UML 模型图编辑器中适当的工具按钮同样可以完成在前节中所叙述的建模设计—编码实现—模型刷新的过程。


ModelMaker 利用模型图视图可以管理任意数量、任意种类的 UML 模型图。每一个 UML 图仅仅包括了模型设计中的某一方面,从而使设计人员更关注于当前设计的建模过程。ModelMaker 中的模型图视图管理了建模过程中用到的所有的 UML 图,甚至可以在 ModelMaker 模型图中建立图元符号与代码模型之间的链接、引用关系。

ModelMaker 的模型图视图同其他视图一样,以类似于资源管理器的形式来组织和管理建模过程中所生成的各种模型图。在视图的模型图列表中以树(层次结构)的形式来组织区分用户建立的模型图,各模型图可以独立命名,模型图名称允许重复。同在资源管理器中一样用户可以使用拖拽、点选技术来实现模型图组织、位置的变换、更名等操作。

注意 删除或清空图元、模型图,并不会删除 ModelMaker 内部模型中所包含的对应数据,而仅仅是从图形化的表示中删除了相应的元素。

2. 绘制 UML 图形工具支持

UML 模型图的绘制在图形编辑器中完成,图形编辑器可以看做是对内部模型的另一观察角度。图形编辑器为每一类图形提供相应的编辑工具条,内含绘制 UML 模型图相应的工具按钮。在后面介绍各种 UML 模型图绘制时,我们将会详细介绍各工具按钮的具体用途。

绘制模型图时都需用到的工具条  各按钮功能如下:

- 选择切换按钮——指针转变为图元选择状态,以便选择当前模型 UML 图中的图元符号。
- 图元删除按钮——指针转换为图元删除状态,以便点选删除当前被编辑模型 UML 图中相应的图元符号,同时在内部模型中对应的实体数据仍然被保留。
- 实体删除按钮——指针转换为实体删除状态,从模型图中删除相应的图元符号,同时从内部模型中删除相对应的实体数据。

在 ModelMaker 支持的各种 UML 图中,图元是构成 UML 图的基本元素。模型图编辑器中包含了创建模型图基本图元的各种工具,这些工具条随 UML 图类型的变化自动调整以方便用户的使用。在 UML 图中图元可以多种形式显示,如矩形、椭圆形或其他形状。其中 UML 图的显示属性(在编辑器的属性设置对话框中设置)和图元的显示属性(在图元属性设置对话框设置)决定了 UML 图、图元在 ModelMaker 中具体是如何显示的,这些属性设置对话框都可以通过上下文相关菜单迅速调出。UML 图中的绝大多数图符都代表了内部模型中的相应元素。

各图元之间的相互关系,在 UML 中以关联来表示,通常为一种双向关系。关联的各图元之间通过关联来索引到对方,各图元之间的关联关系在 UML 各模型图中有其各自特殊的含义,含义具体内容可参考 UML 中对关联的具体定义。关联使相关的图元联合起来共同构筑成 UML 图所表示的含义。各图元之间的关联在 UML 图中代表了特定的含义,在模型图中关联往往都利用一条直线标示,直线的线形以及箭头依照 UML 的定义分别具有特定的含义。关联依靠“锚点”与图元连接,ModelMaker 根据图元的位置以及锚点的情况自动更新相关的图元及关联。

处理关联的相关操作如图 A-9 所示。在很多情况下,尤其是出于美观的考虑,用户可能希望以折线的形式表现图元之间的关联。在 ModelMaker 中只需点选图元之间的关联后,按住 CTRL 键拖动关联线段的转折点就可完成。折线点可以利用右键菜单中的 Delete 选项删除,或直接将折线点拖到两点之间的直线上即可将折线转换为直线。

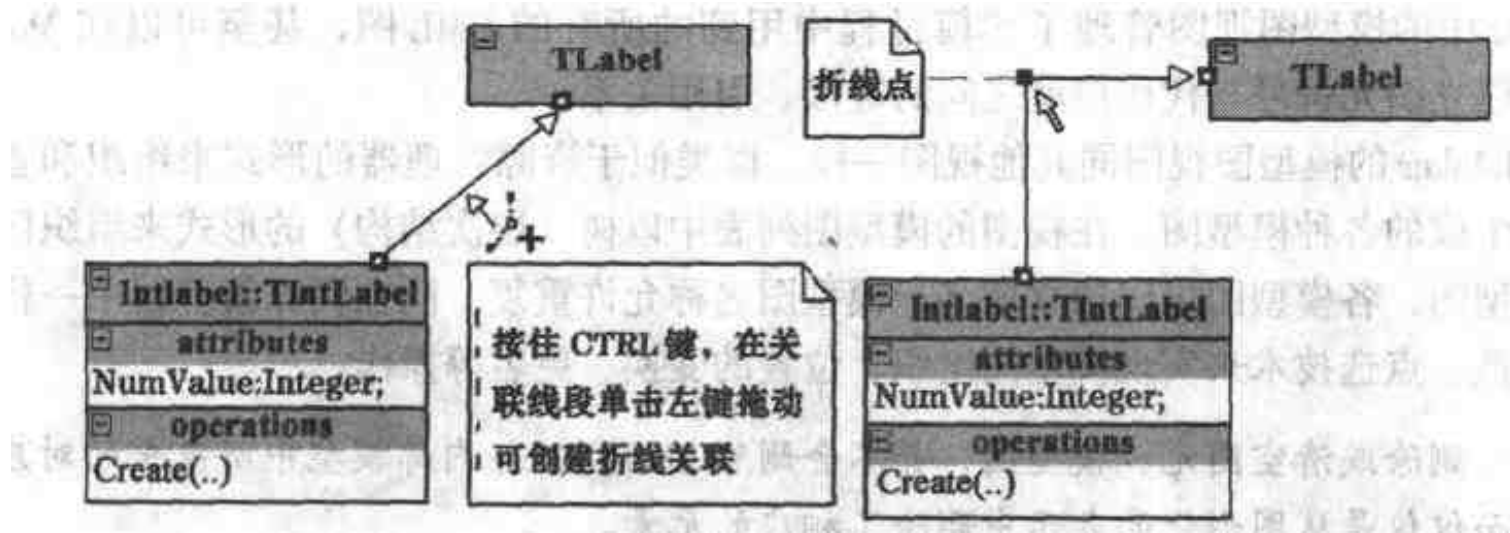


图 A-9 创建折线关联

ModelMaker 图形编辑器提供一些常用的操作供绘制 UML 模型图使用。如:点击图元或关联线选择它们,或用 Shift + Click 将选中的图元添加到当前选择集合中;利用矩形框选图元或关联;双击任何图元或关联调用图元/关联属性编辑对话框设置、修改图元/关联属性。利用鼠标可简单地改变图元的位置、大小,与图元连接的关联的位置、长度也会自动更新。


在图形编辑器中利用拖拽等操作可将类视图/单元视图中列出的类/接口直接转换为相关的 UML 图符,模型中还会根据类/接口之间的相互关系自动绘制相关图元之间的关联关系。这是由 ModelMaker 所提供的引擎自动完成的,虽然很多工作由其自动完成了,但只有开发人员熟练地掌握 UML,才能更好地利用 ModelMaker 工具结合 Delphi 完成软件工程项目的设计、开发过程。

3. UML 图

ModelMaker 支持用例图、类图、序列图、状态图、活动图、协作图、包图、鲁棒图、实现图、思维图等 UML 图形的绘制。ModelMaker 图形编辑器提供一些常用的操作及工具条供绘制各类 UML 图使用。并可通过模型图视图来管理、维护各种 UML 图。

(1) 用例图

用例图利用角色和用例定义并描述了系统(子系统或类)的外部可见行为。图 A-10 是一个用例图例图。用例图由角色、用例及它们之间的相互关系等图符共同组成。角色及用例之间的相互关系利用关联来表示。用例图中的图元很多并不是出现在内部模型中的相应元素,而是一种用于描述系统的图形表示。

UML 用例图从高层描述事情的发生经过,当用例图与其他图形链接时,可以用做顶层导航。利用用例图工具条  按钮可以绘制用例图的工作:

- 添加角色按钮——指针转换为添加角色状态,在 UML 图形编辑器中点击时将会添加新的角色符号。双击该符号将弹出角色符号编辑对话框设置属性。
- 添加用例按钮——指针转换为添加用例状态,在 UML 图形编辑器中点击时将会添加新

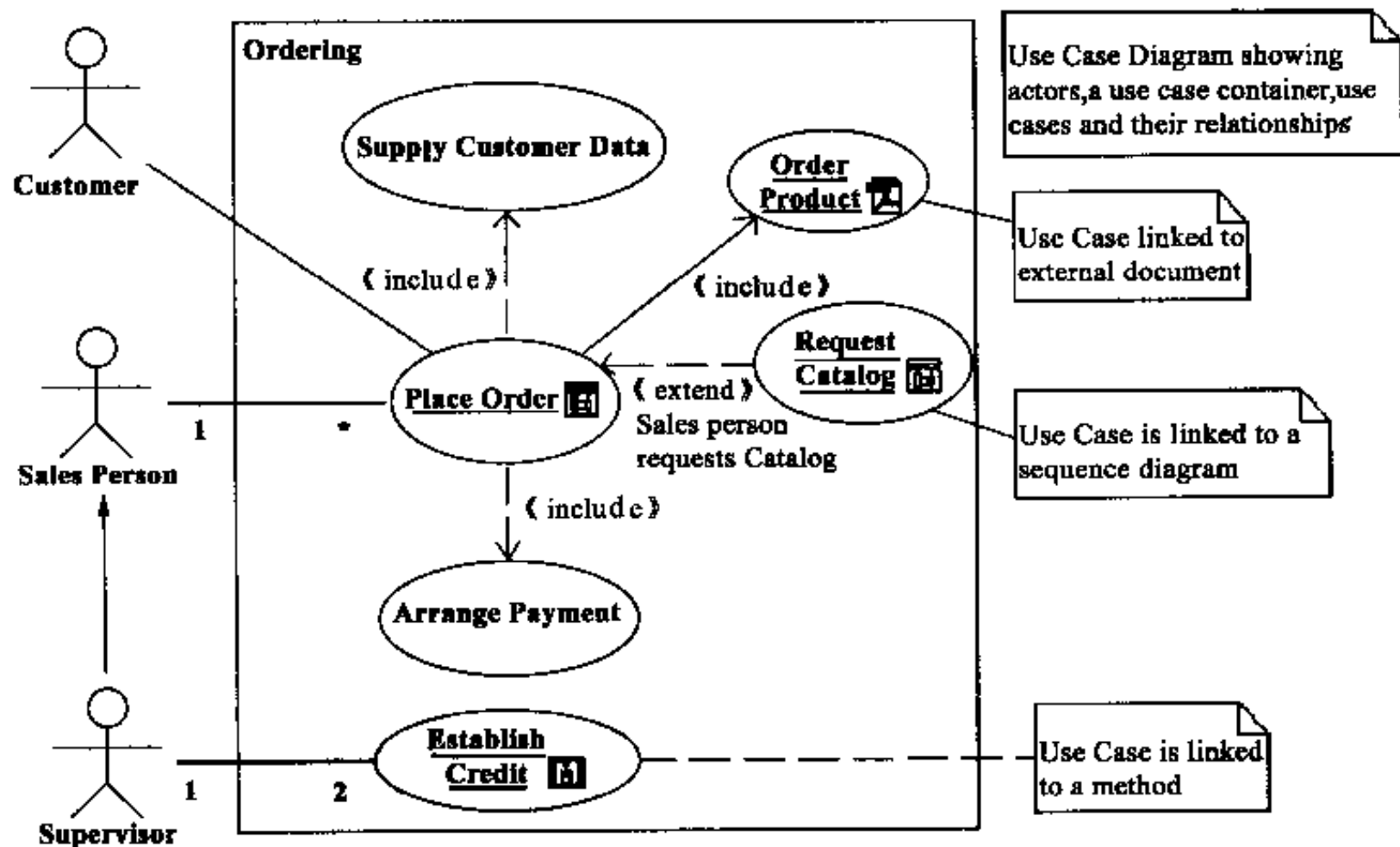


图 A-10 用例图例图

的用例符号。双击该符号将弹出用例符号编辑对话框以设置属性。

- 添加扩展关系按钮——在源用例符号上按下左键后拖动，拖动连接线，至需要建立关联的目的符号后松开左键后将建立源符号与目的符号之间的关联（关联线显示为实线）。输入关联（扩展关系）名称。双击此关联将弹出用例关联编辑对话框以设置属性。
- 添加包含关系按钮——在源用例符号上按下左键后拖动，拖动连接线，至需要建立关联的目的符号后松开左键后将建立源符号与目的符号之间的关联。然后输入关联（包含关系）名称。双击此关联将弹出用例关联编辑对话框以设置属性。
- 添加角色间通信按钮——在一个角色或用例符号上按下左键后拖动，拖动连接线，至需要建立通信的目的角色或用例后松开左键。双击此关联将弹出角色通信编辑对话框以设置属性。
- 添加系统边界按钮——指针转换为添加系统边界状态，点击活动模型图时将会添加新的边界符号。双击该符号将弹出角色边界编辑对话框以设置属性。

(2) 类图

类图提供了模型的静态结构描述，包括在模型中定义的类以及类之间的静态相互关系。在早期设计阶段类图用于捕获问题逻辑方面的内容，在后期设计阶段中类图还可捕获设计决策和实现上的一些细节。图 A-11 展现了利用 ModelMaker 创建的类图例图。

在 ModelMaker 中绘制一个类图时，对于预先编写好的类利用 ModelMaker 的逆向工程系统，可在类视图或单元视图将类直接拖入 UML 图编辑器中。ModelMaker 利用逆向工程引擎可将单元中定义的类、接口、包以及它们之间的相互关系（如继承、组合）等迅速转化为 UML 定义的类型图表示形式。ModelMaker 现在支持 Object Pascal 接口编程，这表明 ModelMaker 提供了一

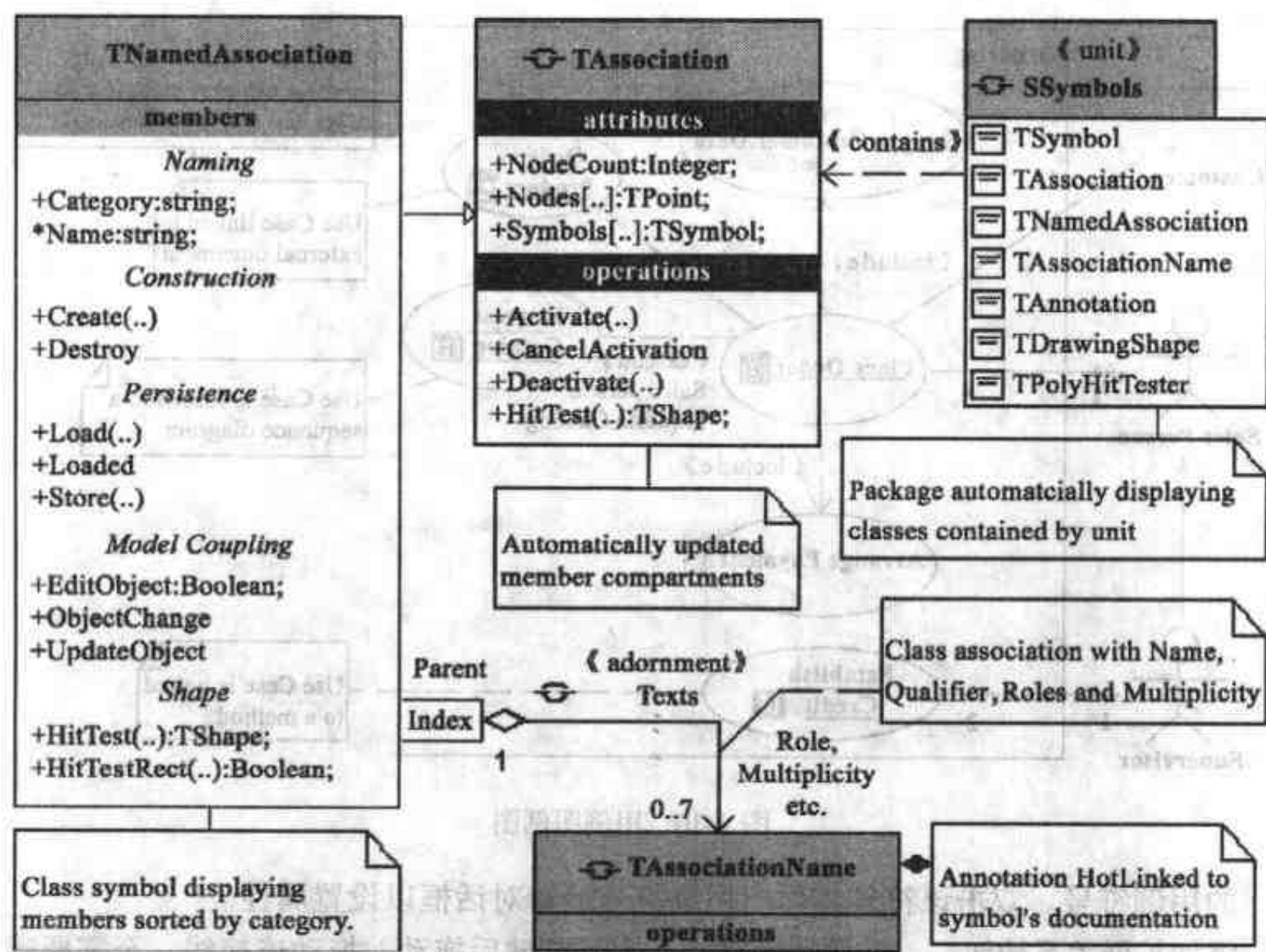


图 A-11 类图例图

种抽象层次上的支持。ModelMaker 还为接口的相关操作提供了向导。

在类图编辑过程中，设计、开发人员可以添加新的类、接口、域、属性与方法，修改类之间的继承关系，甚至可以为内部模型数据中的相关元素添加文档。

使用 ModelMaker 所提供的工具条  可以绘制类图：

- 添加新类按钮——指针转换为添加新类状态，在 UML 图形编辑器中点击时将会在类图中添加新类，显示为类符号；同时会弹出类符号编辑对话框以设置属性。
- 导入已有类按钮——点击此按钮，将显示内部模型中已经建立的类，并给出类列表以供选择。同时弹出类符号编辑对话框以设置导入类的属性。
- 添加新接口按钮——指针转换为添加新接口状态，在 UML 图形编辑器中点击时将会在类图中添加新接口，并在类图中显示为接口符号。同时会弹出接口符号编辑对话框以设置属性。
- 导入已有接口按钮——点击此按钮，将显示已经存在的接口，并给出接口列表以供选择。同时弹出接口符号编辑对话框以设置属性。
- 添加属性按钮——点选按钮后在所要添加属性的类图元上单击鼠标左键，拉连接线到有目标属性类型的类上，或直接释放左键弹出联合属性对话框以创建新的属性。
- 添加域按钮——点选按钮后在所要添加域的类上单击鼠标左键，拉连接线到有目标域


类型的类上，或直接释放左键将弹出联合域创建对话框以创建新的域。

- 添加联合共有类按钮——点选按钮后在一个类或接口上单击鼠标左键，并拉连接线到目标符号，释放左键将弹出联合共有类编辑对话框以设置新增的共有类的属性。
- 添加关联按钮——在源节点上按下左键后拖动，拖动连接线，至需要建立关联的目的节点后松开左键将建立源节点与目的节点之间的关联（关联线显示为实线）。用户双击该连接线显示类图关联编辑对话框，以编辑该关联的属性信息。
- 添加协作按钮——单击按钮以创建一个协作符号。双击此符号，将弹出协作符号编辑对话框以设置属性。
- 添加协作角色关联按钮——在一个协作符号上单击鼠标左键，并拉连接线到目标符号上，释放左键（关联线显示为虚线），输入这个关联的名字。双击该关联将弹出协作角色关联属性编辑对话框以设置属性。
- 添加依赖关联按钮——在源节点上按下左键后拖动，拖动连接线，至需要建立关联的目的节点后松开左键将建立源节点与目的节点之间的关联（关联线显示为虚线）。用户可双击该连接线显示依赖关联编辑对话框，以编辑该关联的属性信息。
- 添加一般关联按钮——在一个源符号上单击鼠标左键，并拉连接线到目标符号上，释放左键。假如已设置该关系，则将进行更新。双击该联合将弹出一般关系编辑对话框以设置属性。
- 添加实现关联按钮——在一个源类符号上单击鼠标左键，并拉连接线到目标接口符号上，释放左键。假如已存在的类接口中没有这个类接口，则将弹出接口向导编辑器以设置属性。

(3) 状态图

每个对象的生命期状态图描述了对象在时间轴上的动态行为，对象被认为是孤立的实体，当事件发生时对象对之响应并与外部对象通信。状态图由状态和状态的迁移共同组成，状态描述了对对象对事件的响应。状态图描述了对象的行为，同时还描述了用例、协作和方法的动态行为，对于对象，状态代表了执行的一个步骤。

状态图是对象的局部化视图，该视图将对象与其生存环境分离，独立地检查它的行为。它是系统行为的缩影，是精确指明对象行为的描述方法。状态图的描述使用对象的状态、事件以及其他状态图所定义的图元符号。

ModelMaker 所提供的工具条  可以方便在图形编辑器中绘制状态图，利用状态图编辑器工具条中的按钮来完成绘制工作。该工具条中的按钮从左至右依次可完成添加对象的状态，添加状态的迁移变化，添加组合状态，添加并发域组合状态，添加域到并发组合状态，添加状态迁移变化，添加初始状态，添加结束状态，添加状态同步，添加动态选择等状态图图元工作。


(4) 序列图

序列图表现的是系统内、外担任不同角色的对象之间依赖于时间轴的动态互动关系，序列图的建立依赖于用例图。一个角色可以是一个类，交互者甚至于是来自系统外的参与者。序列

图中垂直向以角色（或对象）时间轴（生存周期）来组织，序列图就具备了时间顺序概念；序列图水平向表示为不同角色个体之间的交互消息。整个图清晰明确地展示了角色（对象）在其生存周期的任一时刻的动态行为（描述了角色动态行为的时间特性）。

注意 ModelMaker 不能自动地由输入的源代码创建序列图，开发人员只能自己根据系统特性绘制序列图。


通常序列图中的时间顺序是非常重要的，但在实时系统中，时间轴必须是依赖于实时时间的，而对于水平轴上的角色顺序并不在意。

ModelMaker 所提供的工具条  可以方便在图形编辑器中绘制序列图。该工具条按钮从左至右依次可完成以下工作：添加对象实例，添加角色，添加系统边界，添加普通消息响应，添加过程消息响应，创建数据成员/方法/属性/消息，添加异步消息响应，添加反馈消息等。

（5）协作图

协作图以图或者网格的形式展示对象之间的交互关系，着重于描述对象之间交互的空间特性。协作图将所有参与交互的对象绘制于图上，对象之间的协作情况使用消息流来表示。协作图既描述了系统的静态结构，同时也描述了系统的动态行为。

协作图与序列图最大的不同在于：协作图中的对象可以任意布局，序列图中的对象必须水平布局；协作图中的对象之间既可以是相互关联的也可以是依靠消息连接，而序列图中则为消息连接。利用协作图有时可以更清楚地展示对象之间的相互关系。

ModelMaker 所提供的工具条  可以方便在图形编辑器中绘制协作图。其中前 5 个按钮我们前面已非常熟悉，这里不做叙述。其他按钮依次为：添加通讯关联、添加数据成员/属性/方法关联、添加消息（普通/过程/反馈消息等）、添加数据成员/属性/方法交互、添加交互消息（信号量/任务/队列/邮箱等）按钮。这些按钮相互协作可完成协作图的绘制。

（6）活动图

活动图是状态图的一种针对系统 workflow 进行建模分析的特殊表达形式，它通过建立运算到运算的控制流模型来阐明系统的动态本质。活动图的状态代表了运算执行的状态，而非一般对象的状态。

活动图不显示所有运算的细节，也不显示执行活动的对象，只关心活动的工作流。活动图是设计的一个起点，为了完成设计，每个活动必须被扩展成一个或多个的操作，每个操作被指派给特定的对象来实现。活动图通常用于建立 workflow 或业务处理的内部操作模型。

（7）包图

为确保团队中的工作不相互影响，设计开发人员在某一时刻只处理有限的信息，大系统必须被划分为较小的单元部分。包图包含了包和包之间的依赖关系，显示了 Object Pascal 单元之间的相互引用关系。利用“单元依赖性分析”工具的输出可以立即得到分析结果。这种模型图不是 UML 定义的，但 ModelMaker 直接支持。包图将系统的单元以包中相互关联的模块组织起来。

(8) 实现图

许多系统的设计模型独立于最终的实现结构，设计模型主要考虑的是系统的逻辑结构及其设计。而系统最终在重用性和性能考虑上是非常重要的。UML 包括了两种模型图来表现：组件图和部署图。

将可重用的系统代码段封装成可替代的单元，称为组件，组件是用于构造系统的高层次的可重用代码段。实现图以组件、接口以及组件间的依赖关系来阐明如何实现设计元素（如类）。

部署图显示了运行时资源的物理分布。在运行时，结点容纳组件和对象，组件和对象在结点上的分布可以是静态的，也可是动态的，如果组件实例依赖于放置的结点，则部署图可以显示系统在性能上的瓶颈所在。

(9) 鲁棒图

鲁棒图在建模过程中提供用例分析层面与序列图中所显示细节部分的关联。鲁棒性分析提供了一种检测用例是否正确以及系统响应是否有效的方法，它有效地解决了用例图与序列图之间的空白带。

(10) 思维图

思维图（见图 A-12）由围绕系统设计核心思想的若干节点构成，并围绕这些节点记录若干相关的发散性的设计思路。虽然思维图不是 UML 所定义的，但思维图可以让开发人员立即产生几乎无穷的想法，并在极短的时间内按照其相关性组织起来，这样就将设计人员的最初想法保存了起来，接下来就是阅读思维图为每一节点编写详细内容了。

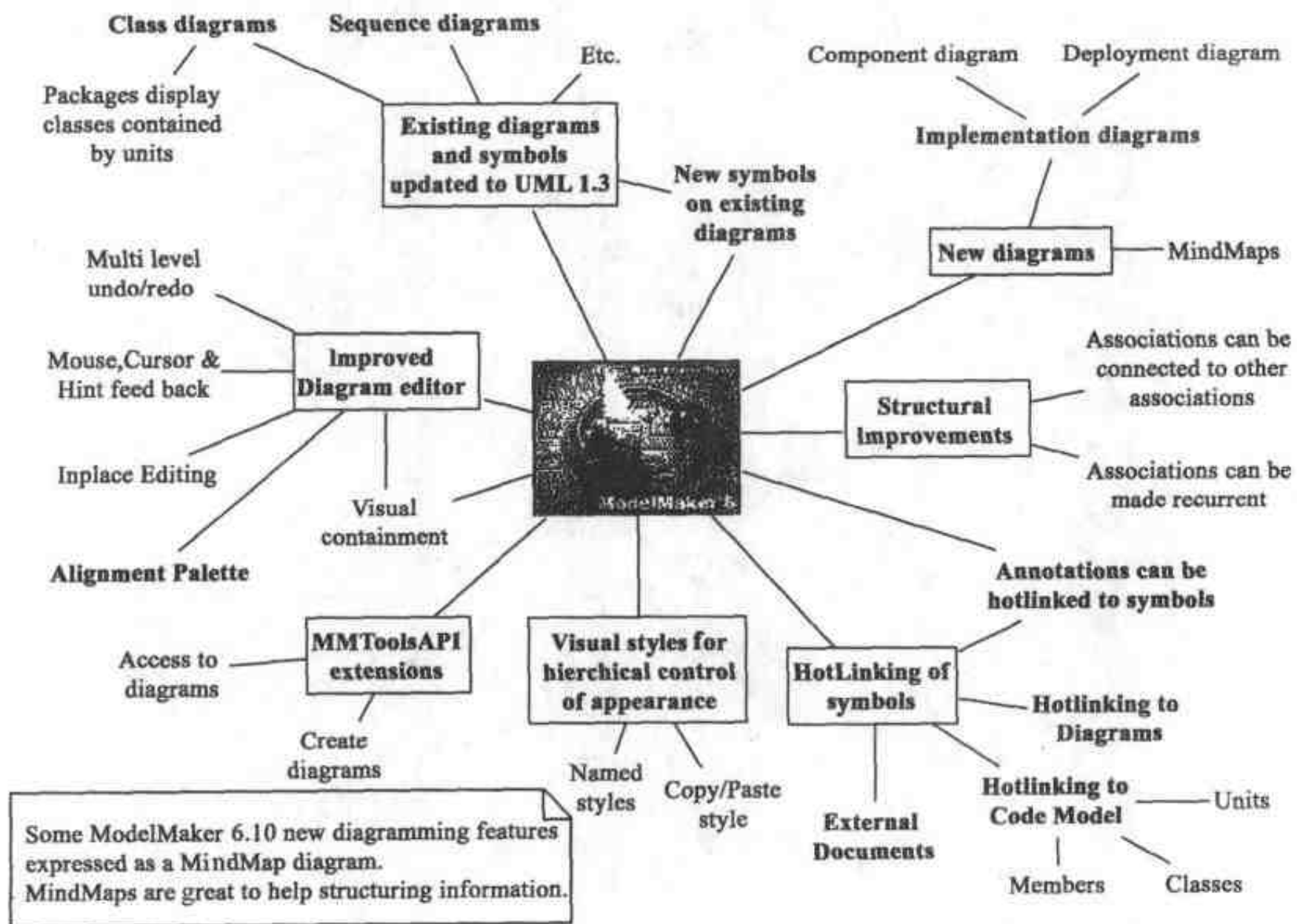


图 A-12 思维图例图

A.3 小结

本文重点从类设计与代码生成、UML 建模两方面介绍了 ModelMaker 作为 Delphi 专用的面向对象建模工具的强大功能和使用方法。Delphi 结合 ModelMaker 将面向对象建模、设计、编码融为一体,再加上强大的 RAD 功能,再次成为最具竞争力和生产效率的开发工具。当然,就像要用好 Delphi 必须掌握面向对象编程思想一样,要玩转 ModelMaker 则必须具备 UML 的良好基础。另外,除了提供 Delphi 7 专用的 ModelMaker 6.2 外,ModelMaker 还有支持低版本 Delphi 的版本,有兴趣的读者可以关注 ModelMaker 网站:<http://www.modelmakertools.com/>。

参考文献

Delphi

- 1 刘艺著·Delphi 6 企业级解决方案及应用剖析·北京：机械工业出版社，2002
- 2 (美)Paul Kimmel 著·Building Delphi 6 Applications·USA: McGraw-Hill Co., 2001
- 3 (美)Ray Lischner 著·Delphi in a Nutshell·USA: O'Reilly & Associates, Inc, 2000
- 4 (美)Steve Teixeira 等著·Delphi 6 开发人员指南·龙劲松等译·北京：机械工业出版社，2003
- 5 李维著·Delphi 5.x ADO/MTS/COM + 高级程序设计篇·北京：机械工业出版社，2000
- 6 李维著·Delphi 5.x 分布式多层应用系统篇·北京：机械工业出版社，2000
- 7 李维著·Delphi 6/Kylix2 SOAP/Web Service 程序设计篇·北京：机械工业出版社，2002

其他语言

- 8 (美)Bruce Eckel 著·Thinking in Java(SE)·北京：机械工业出版社，2002
- 9 (美)Klaus Michelsen 著·C# Primer Plus·USA: Sams Publishing, 2002
- 10 (美)Eric Armstrong 著·JBuilder 2 Bible·USA: IDG Books Worldwide, Inc, 1998
- 11 阎宏著·Java 与模式·北京：电子工业出版社，2002
- 12 (美)Sten Sundblad 等著·Windows DNA 可扩展设计·前导工作室译·北京：机械工业出版社，2001
- 13 (美)Richard C. Lee 等著·C++ 面向对象开发·麻志毅等译·北京：机械工业出版社，2002

软件工程

- 14 (美)Erich Gamma 等著·Design Patterns: Elements of Reusable Object-Oriented Software·USA: Addison Wesley Longman, Inc, 1995
- 15 (美)Ian Graham 著·面向对象方法：原理与实践·第3版·袁兆山等译·北京：机械工业出版社，2003
- 16 (美)Roger S. Pressman 著·软件工程：实践者的研究方法·第5版·梅宏译·北京：机械工业出版社，2002
- 17 (英)Ian Sommerville 著·Software Engineering·USA: Addison-Wesley Publishers Ltd., 2001
- 18 (美)James Rumbaugh, Ivar Jacobson, Grady Booch 著·The Unified Modeling Language Reference Manual·USA: Addison Wesley Longman, Inc, 1999
- 19 (美)Sinan Si Albir 著·UML 技术手册·常晓波译·北京：中国电力出版社，2002

来自互联网上的文献资料

- 20 虫虫·天方夜谭 VCL·C++ View, 2001
- 21 Code64·Building Web Application With IntraWeb, 2002
- 22 (意)Marco Cantu·Comparing OOP Languages: Java, C++ , Object Pascal, 1997
- 23 (美)Mark Miller·Reuse through Inheritance and Object Composition: Good Class Design in Delphi, 1998
- 24 (国籍不详)Charles E. Weindorf·Delphi Databases: Dynamic Datamodules at Runtime

[G e n e r a l I n f o r m a t i o n]

书名 = D e l p h i 面向对象编程思想

作者 =

页数 = 4 7 6

S S 号 = 1 1 1 1 7 9 8 2

出版日期 =